TRANSPUTER
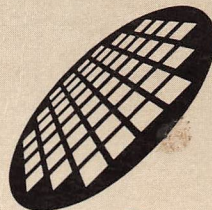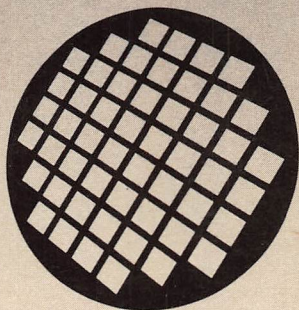
TECHNICAL

NOTES

TRANSPUTER
TECHNICAL NOTES

Other titles in this series
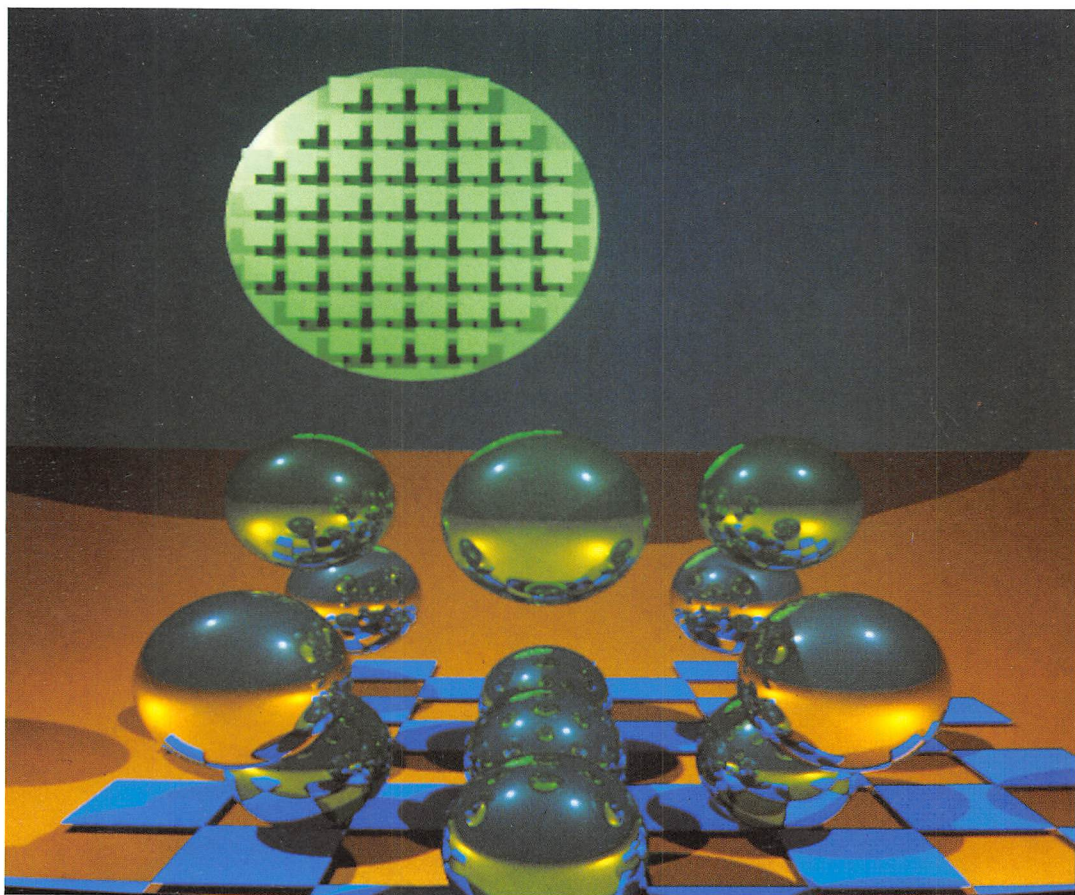
Transputer Reference Manual
Transputer Development System
Communicating Process Architecture
Transputer Technical Notes
Transputer Instruction Set: a compiler writer's guide
Digital Signal Processing

Ray traced image with anti-aliasing

TRANSPUTER
TECHNICAL NOTES

INMOS Limited

Prentice Hall
New York London Toronto Sydney Tokyo

Contents

2   Systems

Contents overview

Hardware

1	*Designing with the IMS T414 and IMS T800 memory interface*	Explains the use of the transputer memory interface. *(INMOS technical note 09)*
2	*Connecting INMOS links*	Discusses how links are used for local and long distance communication. *(INMOS technical note 18)*
3	*IMS B003 design of a multi-transputer board*	Describes the design of a simple multiple transputer board. *(INMOS technical note 10)*

Systems

4	*Designs and applications for the IMS C004*	Explains how the IMS C004 link switch can construct a number of different networks. *(INMOS technical note 19)*
5	*Module motherboard architecture*	Describes the architecture of a range of transputer modules and motherboards. *(INMOS technical note 49)*
6	*Dual inline transputer modules (TRAMs)*	Provides a detailed specification of the transputer modules. *(INMOS technical note 29)*

Software

7	*Program design for concurrent sytems*	Discusses the design of concurrent processing systems. *(INMOS technical note 05)*
8	*Exploring multiple transputer arrays*	Explains how a transputer array can be explored and tested by a 'worm' program. *(INMOS technical note 24)*
9	*Extraordinary use of transputer links*	Describes the software needed to recover from failure of communication via a transputer link. *(INMOS technical note 01)*

Applications

10	*A transputer based radio-navigation system*	Involves the analysis of incoming radio signals in real time. *(INMOS technical note 00)*
11	*The transputer based navigation system – testing embedded systems*	Discusses system integration and testing using the navigation system as an example. *(INMOS technical note 02)*
12	*A transputer based distributed graphics display*	Describes a high performance graphics system showing the performance requirements of the common graphics operations. *(INMOS technical note 46)*

Performance

13	*Lies, damned lies and benchmarks*	Explains the performance of the transputer measured by the standard Whetstone, Dhrystone and Savage benchmarks. *(INMOS technical note 27)*
14	*Performance maximisation*	Describes the techniques for optimising system performance. *(INMOS technical note 17)*

Preface

Transputer Technical Notes is a collection of classic papers written by INMOS engineers to assist in the implementation and development of transputer technology. The collection is presented in hardware, systems, software, applications and performance sections which combine together to describe an approach to system design and development using transputer technology.

The papers were originally written as a series of individual technical notes each intended to expand and develop a particular area of interest or application. The collection will be of interest to electronic engineers, software engineers, programmers, system designers and managers. It has been published in response to the growing interest and requests for information concerning the transputer and occam.

The INMOS transputer family is a range of VLSI building blocks for concurrent processing systems, with occam as the associated design formalism. occam is an easy and natural language for the programming and specification of concurrent systems. Further information explaining the architectural foundation of occam and the INMOS transputer is included in a similar collection of classic papers, entitled 'Communicating Process Architecture'.

Current INMOS transputer products include the 16 bit IMS T212 and IMS T222, the 32 bit IMS T414 and IMS T425, and the IMS T800, a 32 bit transputer with an integral high speed floating point processor. The transputer is fully supported by INMOS development tools and standard language compilers. The family also includes peripheral controllers and communications products. Detailed information describing individual devices is available in the 'Transputer Reference Manual'.

The IMS M212 is an intelligent peripheral controller comprising a 16 bit processor, on chip memory and communications links. It contains hardware and interface logic to control disc drives, and can be used as a programmable disc controller or as a general purpose peripheral interface.

The INMOS serial communication link is a high speed system interconnect which provides full duplex communication between members of the transputer family. It can be used as a general purpose interconnect even where transputers are not used. The IMS C011 and IMS C012 link adaptors are communications devices enabling the INMOS serial communication link to be connected to parallel data ports and microprocessor buses. The IMS C004 is a programmable link switch. It provides a full crossbar switch between 32 link inputs and 32 link outputs.

The Transputer Development System referred to in this manual comprises an integrated editor, compiler and debugging system which enables transputers to be programmed in occam and in industry standard languages. Detailed information describing the Transputer Development System is available in the 'Transputer Development System' manual.
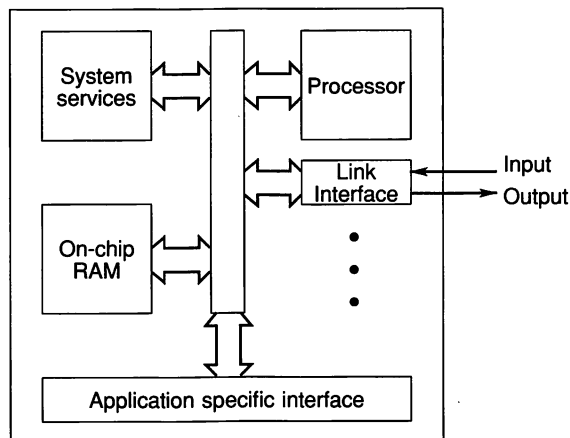
Transputer architecture

Introduction

This book, in five sections, is a compilation of INMOS technical notes describing an approach to system design and development using transputer technology.

The first section, *hardware*, explains the use of the transputer interface and describes a simple multiple transputer board. The section begins by explaining the use of the transputer memory interface. Memory system design is discussed and the choice of memory interface configuration and timing is explained. Examples of static and dynamic RAM systems are included. The section continues with an explanation of the use of INMOS links. Links are normally used for local communication on a circuit board or within a cabinet. A detailed explanation of several techniques for long distance communication using links is also provided. The section concludes with a complete example of a simple multiple transputer board.

The second section, *systems*, explains the use of transputers and link switches to construct modular systems in which the link configuration can be programmed. The section begins by showing how the IMS C004 link switch can be used to construct a number of different networks. A description of the switch is supplied as an occam program and a formal specification of the switch given in the CSP notation. The section continues with a description of the modular transputer system. This consists of a range of transputer modules which can be plugged into motherboards. Each motherboard is fitted with IMS C004 configuration switches. The section concludes with a detailed description of the transputer modules.

The third section, *software*, includes software topics of interest when constructing multiple transputer systems. The section begins with a discussion of the design of concurrent processing systems. The use of occam and the Transputer Development System is outlined and a simple example of system integration and testing is shown. The section continues with two more specialised topics. The first explains how a transputer array can be explored and tested using a special 'worm' program. The program gradually spreads through the network testing the transputers and constructing a map of the network as it passes through them. The section concludes with the second topic, a description of the software needed to recover from failure of communication via a transputer link. Failure arising from electrical interference, disconnection and incorrect behaviour of the remote transputer are all considered and use of the appropriate occam procedures explained.

The fourth section, *applications*, features two applications of transputers. The first application is a radio navigation system and involves the analysis of incoming radio signals in real time. The design of the system, using occam, is described in detail. A discussion follows describing system integration and testing using the navigation system as an example. The second application is a high performance graphics system. An introduction to computer graphics is followed by a discussion of the architecture of graphics systems including performance requirements of the common graphics operations. Finally, a detailed description of the design of a modular transputer based graphics system is provided.

The final section, *performance*, examines the performance of transputers and includes some techniques for maximising performance. The section begins with a detailed explanation of the performance of the transputers measured by the standard Whetstone, Dhrystone, and Savage benchmarks. Comparisons with other computers and microcomputers are made. The section continues with a description of the techniques for optimising performance. Examples of optimised sequential programs are shown, followed by a discussion of the optimisation of concurrent systems. The section concludes with an example of an optimised graphics program.

Hardware

1 Designing with the IMS T414 and IMS T800 memory interface

1.1 Overview of the memory interface

The IMS T414 and IMS T800 have a configurable memory interface designed to allow easy interfacing of a variety of memory types with a minimum of extra components. The interface can directly support DRAMs, SRAMs, ROMs and memory mapped peripherals. The interface is the same for both parts so for 'T414' read 'T414 and T800' throughout.

The T414 has a 32 bit multiplexed data and address bus with a linear address space of 4 Gbytes. There are 4 byte write strobes, a read strobe, a refresh strobe, 5 configurable strobes, a wait input, a memory configuration input, a bus request input and bus grant output. Figure 1.1 shows the inputs and outputs for the T414 transputer that are associated with the memory interface.

notMemWrB0–3	4 → byte write strobes
notMemRd	read strobe
notMemRf	refresh strobe
notMemS0–4	5 → configurable strobes
MemnotWrD0	notWriteFlag/data 0
MemnotRfD1	notRefreshFlag/data 1
MemAD2–31	30 → address/data 2–31
MemReq	external request
MemGranted	external request granted
MemWait	wait states
MemConfig	configuration input
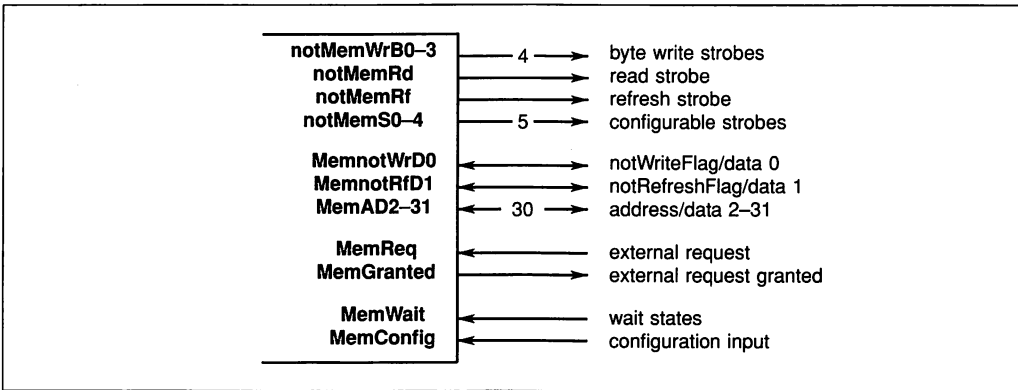
Figure 1.1

With this flexible arrangement, a variety of memory timing controls can be obtained with little external hardware. An example of bus timing is shown in figure 1.2.
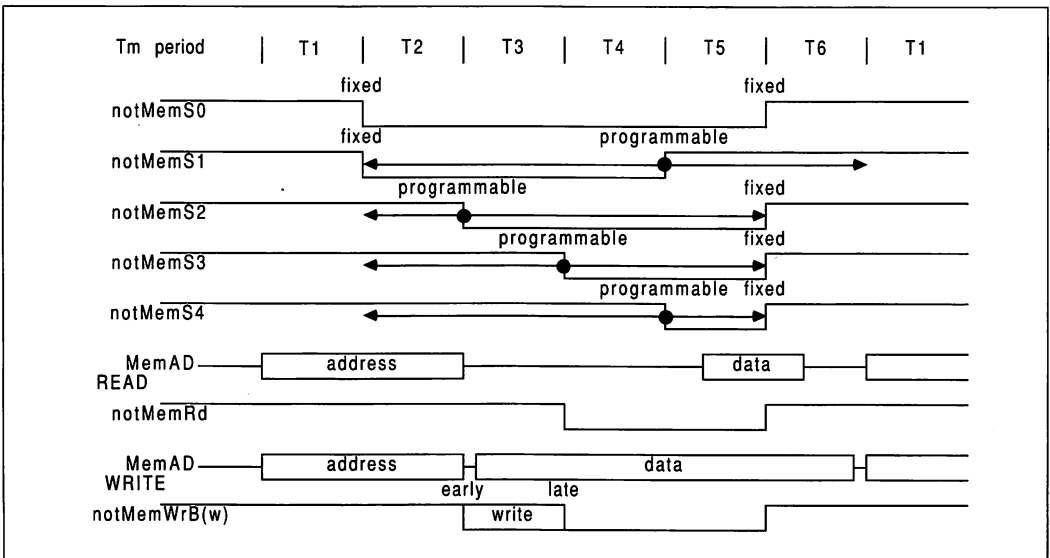
Figure 1.2

The T414 has a signed address space and addresses memory as bytes. Addresses, therefore, run from $80000000 through $FFFFFFFF to $7FFFFFFF. This differs from the occam map which starts at $0 and is organised as words. The comparison, for the T414, is given in figure 1.3: the T800 has MemStart at $80000070 and start of external memory at $80001000.

Machine Map		Byte address	Word offsets	Occam Map
hi	lo			
Reset Inst		#7FFFFFFE (ResetCodePtr)		
Memory configuration		#7FFFFFF8 to #7FFFFF6C		
		#0		
		#80000800 - Start of external memory - #0200		
		#80000048 **MemStart**	**MemStart** #12	
Processor use				Processor use
Event		#80000020	#08	Event
Link 3 Input		#8000001C	#07	Link 3 Input
Link 2 Input		#80000018	#06	Link 2 Input
Link 1 Input		#80000014	#05	Link 1 Input
Link 0 Input		#80000010	#04	Link 0 Input
Link 3 Output		#8000000C	#03	Link 3 Output
Link 2 Output		#80000008	#02	Link 2 Output
Link 1 Output		#80000004	#01	Link 1 Output
Link 0 Output		#80000000	#00	Link 0 Output
		(MOSTNEG INT)	(Base of memory)	

Figure 1.3

Throughout this application note, all addresses referred to will be those for the machine map.

The T414 has 2Kbytes of on-chip RAM at addresses $80000000 to $800007FF: the T800 has 4Kbytes at addresses $80000000 to $80000FFF. It is, therefore, advisable for $80000000 to $FFFFFFFF to be used for RAM and $00000000 to $7FFFFFFF to be used for ROM and I/O. If internal memory and external memory exist at the same address, the transputer will access internal memory. Note that if the memory map is not completely decoded, it is usually possible to access the 'hidden' external memory at another address; e.g. on the B004-2, the hidden memory can actually be accessed at $80200000 to $802007FF.

1.1.1   Memory interface timing

The T414 memory interface cycle has six timing states, referred to as **Tstates**. The **Tstates** have the nominal functions:

Tstate

T1	address setup time before address valid strobe
T2	address hold time after address valid strobe
T3	read cycle tristate/write cycle data setup
T4	extended for wait states
T5	read or write data
T6	end tristate/data hold

The duration of each **Tstate** is configurable to suit the memory devices used and can be from one to four **Tm** periods. One **Tm** period is half the processor cycle time; i.e. half the period of **ProcClockOut**. Thus, **Tm** is

25nsec for a T414-20 (20MHz transputer). **T4** may be extended by wait states in the form of additional **Tms**. **A0** and **A1** are not output with the rest of the address. During a write cycle, byte and half-word (16 bit data) addressing is achieved by the four write byte strobes (**notMemWrB**): only the write strobes corresponding to the bytes to be written are active. During a read cycle, this is achieved by internally selecting the bytes to be read.

Thus, the two lowest order address lines are not needed. However, care must be taken when mapping byte wide peripherals onto the interface, as they will have to be addressed on word boundaries.

The two lowest order data lines are not multiplexed with address lines but, during the address period, are used to give early indication of the type of cycle which will follow:

> **MemnotWrD0** is low during **T1** and **T2** of a write cycle.

> **MemnotRfD1** is low during **T1** and **T2** of a refresh cycle.

The use of the strobes notMemS0 to notMemS4 will depend upon the memory system. The rising edge of **notMemS1** and the falling edges of **notMemS2** to **notMemS4** can be configured to occur from 1 to 31 Tm periods after the start of T2. This is summarised in figure 1.2 and in the table below.

Signal	Starts	Ends	
notMemS0	T2	T6	
notMemS1	T2	T2 + (Tm*s1)	(or end of T6 if this occurs first)
notMemS2	T2 + (Tm*s2)	T6	
notMemS3	T2 + (Tm*s3)	T6	
notMemS4	T2 + (Tm*s4)	T6	

It should be noted that the use of wait states can advance the rising edge of notMemS1 in relation to that of the other strobes; care must be taken if this signal is being used for RAS driving DRAMs for which RAS must not be removed before CAS.

1.1.2   Early and late write

The **notMemWrB** strobes can be configured to fall either at the beginning of **T3** (early write) or at the beginning of **T4** (late write); the rising edge is always at the beginning of **T6**. Early write gives a longer set up time for the write strobe but data is only valid on the rising edge of the pulse. For late write, data is also valid on the falling edge of the strobe but the pulse is shorter.

1.1.3   Refresh

The T414 has an on-chip refresh controller and 10 bit refresh address counter and can, therefore, refresh DRAMs of up to 1Mbit by 1 capacity without requiring the counter to be extended externally.

Refresh can be configured to be either enabled or disabled. If enabled, the refresh interval can be configured to be 18, 36, 54 or 72 **ClockIn** periods; though if a refresh cycle is due, the current memory cycle is always completed first. The time between refresh cycles is thus almost independant of transputer speed and the length of memory cycles.

Refresh cycles are flagged by **notMemRf** going low before T1 and remaining low until the end of T6. Refresh is also indicated by **MemnotRfD1** going low during T1 and T2 with the same timing as address signals. The address output during refresh is:

AD0	= MemnotWrD0	high
AD1	= MemnotRfD1	low, to indicate refresh
AD2 – AD11		refresh address
AD12 – AD30		high
AD31		low

During refresh cycles, the strobes **notMemS0 – notMemS4** are generated as normal.

1.1.4   Wait states and extra cycles

Memory cycles can be extended by wait states. **MemWait** is sampled close to the falling edge of **ProcClock-Out** prior to, but not at, the end of **T4**. If it is high, **T4** is extended by additional **Tm**s (shown as "W" by the memory interface program). Wait states are inserted for as long as **MemWait** is held high, **T5** proceeds when **MemWait** is low. Note that the internal logic of the memory interface ensures that, if wait states are inserted, **T5** always begins on a rising edge of **ProcClockOut**: so the number of wait states inserted will be either always odd or always even, depending on the memory configuration being used.

Every memory interface cycle must consist of a number of complete cycles of **ProcClockOut**: i.e. it must consist of an even number of **Tm**s. If there are an odd number of **Tm** periods up to and including **T6**, an extra **Tm** (shown as 'E' by the memory interface program) will be inserted after **T6**.

1.1.5   Setting the memory interface configuration

A memory interface configuration is specified by a 36 bit word and is fixed at reset time. The T414 has a selection of 13 pre-programmed configurations. If none of these is suitable, a different configuration can be selected by supplying the complement of the configuration word to the T414s **MemConfig** input immediately following reset.

A pre-programmed configuration is selected by connecting **MemConfig** to **MemnotWrD0**, **MemnotRfD1**, **MemAD2–MemAD11** or **MemAD31**. Immediately after reset, the T414 takes all of the data lines high and then, beginning with **MemnotWrD0**, they are taken low in sequence. If **MemConfig** goes low when the T414 pulls a particular data line low, the memory interface configuration associated with that data line is used. If, during the scan, **MemConfig** is held low until **MemnotWrD0** goes low, or is connected to **MemAD31**, the slowest memory configuration is used.

After scanning the data lines as described above, the T414 performs 36 read cycles from locations $7FFFFF6C, $7FFFFF70 – $7FFFFFF8. No data is latched off the data bus but, if **MemConfig** was held low until **MemnotWrD0** was taken low, each read cycle latches one bit of the (inverted) configuration word on **MemConfig**. Thus, a memory configuration can be supplied by external logic.

Using a pre-programmed configuration has the advantage of requiring no external components: only a connection from **MemConfig** to the appropriate data line. However, selecting an external configuration can also be very economical in component use. If the transputer is booting from ROM, the ROM must occupy the top of the address space. One bit of the memory configuration word can be stored in each of the 36 addresses mentioned above and the only additional hardware required is an inverter connecting the appropriate data line (usually **MemnotWrD0**) to **MemConfig**. **MemConfig** is thus held low until **MemnotWrD0** goes low and is fed with the inverse of the configuration word during the 36 read cycles. Alternatively, the inverted configuration word can be generated from **A2–A7** by one sum term of a PAL.

1.1.6   The memory interface program

The INMOS Transputer Development System includes an interactive program which assists in the task of memory interface design. The program produces timing diagrams and timing information so that the designer can see the effects of varying the length of each **Tstate** and the positions of the programmable strobe edges. Of course, the program cannot allow for external logic delays and loading effects as these are system dependant but it does assist greatly in preliminary design.

1.2   Basic considerations in memory design

1.2.1   Minimum memory interface cycle time

The minimum number of processor clock cycles for an external memory access is 3, which occurs when all **Tstates** are 1 **Tm**. With a 50 nsec cycle time, this will be 150 nsec.

The most important DRAM parameters to be considered at the start of a memory design are the access and cycle times and the RAS precharge time. These will be a guide to the fastest timing possible, which is generally a good starting point, and are defined in figure 1.4.
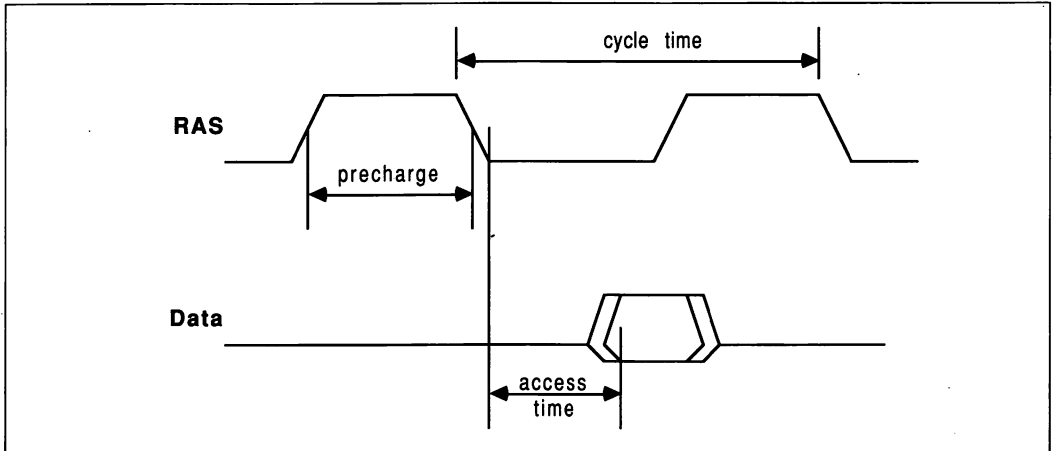
Figure 1.4

Parameters for typical Dynamic RAMS:

	NEC uPD41256-15	NEC uPD41256-12	Hitachi HM51256-10
Access time	150ns	120ns	100ns
Cycle time	260ns	220ns	180ns
RAS precharge	100ns	90ns	70ns

	NMB AAA2800-150	AAA2800-80
Access time	150ns	80ns
Cycle time	246ns	151ns
RAS precharge	90ns	65ns

Higher density devices require longer RAS precharge times but, if the memory does not require RAS to remain low until the end of the memory cycle, it can be taken high before the cycle ends, thus easing the designer's job of finding adequate precharge time whilst minimising the amount of time to be added to the DRAM cycle time.

1.2.2   Delay and skew

When calculating memory interface timings, consideration must be given to propagation delay and skew through buffers and decoding. Skew occurs where there are different logic thresholds and hence different propagation delays for high going and low going signals. This is shown in figure 1.5.

It is also important to bear in mind the asymmetric drive capabilities of most logic that would be used externally.
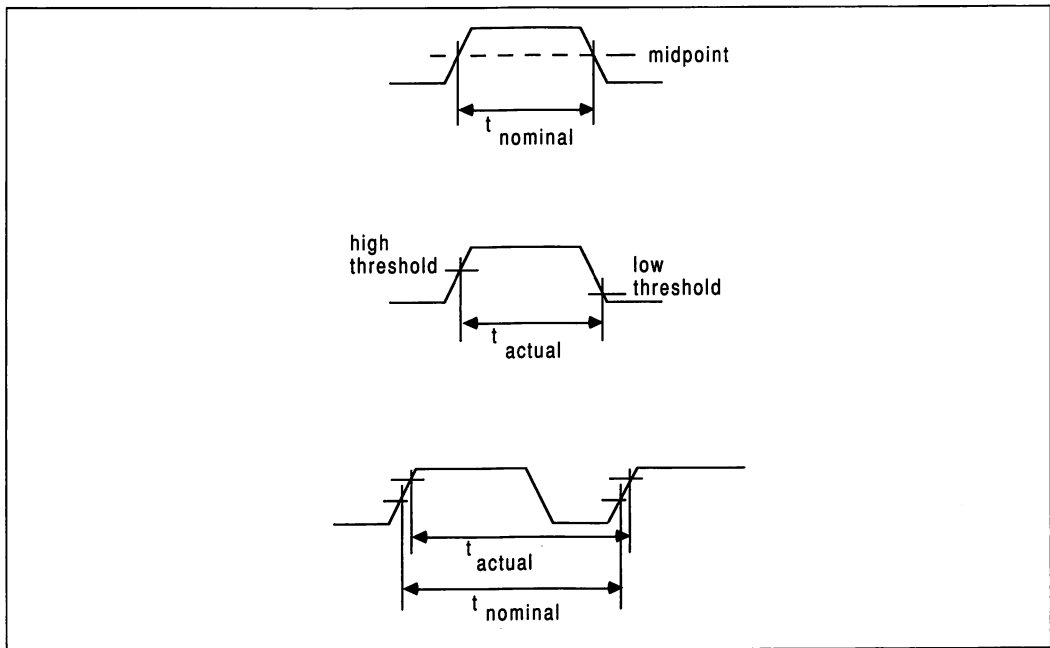
Figure 1.5

1.2.3   Ringing

Ringing (figure 1.6) becomes a problem when signals are called upon to drive a large capacitive load, such as a DRAM array. The high currents required to charge the capacitance have to flow through wiring or PCB tracks, all of which have some inductance, thus creating a tuned circuit. Ideally, the waveform presented will be as steep as possible for minimum propagation delays; however, this implies a large spread of frequencies, including the resonant frequency of the tuned circuit. An alternative way to view the problem is that of driving a transmission line. The solution is to include a series resistor to dissipate the energy in the tuned circuit whilst matching the driver more closely to the transmission line characteristic impedance. The aim is critical damping of the response to the step input. Some DRAM buffers/drivers have the series resistor, or something equivalent, incorporated. e.g. AMD Am2965/6.
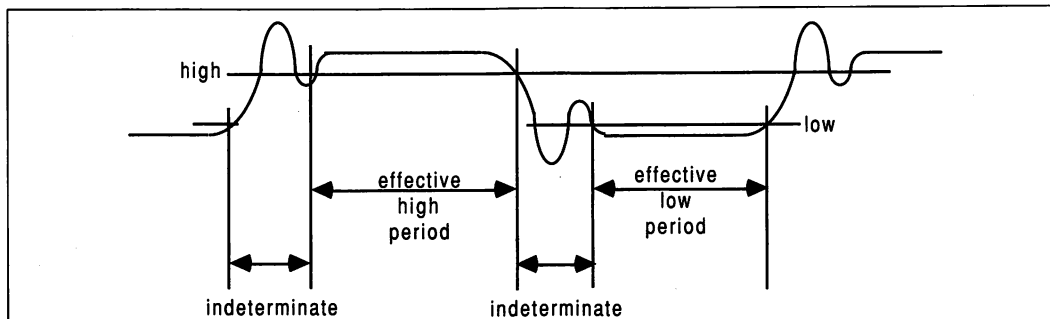
Figure 1.6

1.3 Worked example

This example describes the design of a system based on a T414-20 with:

1 2 Mbytes of RAM.

2 A 1 Mbyte ROM space.

3 A 1 Mbyte I/O space.

Warning: A number of common pitfalls exist in this application, and are revealed step by step. Thus the partial circuits should not be used until this complete section has been read and digested.

1.3.1 Choose memory device size

The most compact way to implement the 2 Mbyte memory is as two banks of 256k x 1 bit DRAMs. This requires 64 devices.

1.3.2 Choose RAS duty cycle

A T414-20 has been specified as the design goal. This gives a **Tm** period of 25 nsec. To run as fast as possible, let **T1 – T6** each be 1 **Tm** in length; giving an external memory cycle time of 150 nsec. Such a short memory cycle time requires the use of a fast, high performance DRAM.

With only 3 processor cycles, there is only one realistic possibility, as shown in figure 1.7, namely RAS low for three **Tm** periods. RAS low for two **Tm** periods would require a 50 nsec access DRAM and RAS low for four **Tm** periods leaves only 50 nsec for RAS precharge. Neither of these is possible with current DRAMs.
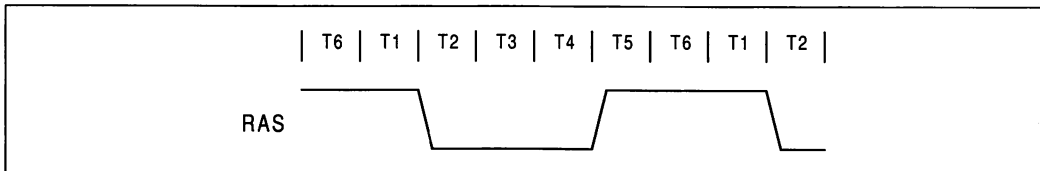
Figure 1.7

1.3.3 Allocate strobes

Most current EPROMs and peripherals cannot run at a cycle time of 150 nsec. The fastest widely available EPROMs are 150nsec access. Thus it will be necessary to insert wait states when EPROMs and peripherals are accessed. To maximise the system performance it will be necessary to have two different lengths of wait states, one for ROM and one for peripherals, requiring the use of two of the transputer's programmable strobes. This means that only a change to the memory configuration will be required at a later date to upgrade to faster parts. Therefore, we will reserve **notMemS3** and **notMemS4** as two separate wait state generators, since the point at which they go low is the feature that is user programmable.

This leaves 3 strobes, **notMemS0-2** for total DRAM control.

notMemS0 goes low at the start of **T2** and high at the start of **T6**, being low for 4 **Tm** periods in this example, and thus cannot be used for RAS. The data and address lines from the transputer are multiplexed, addresses being valid for **T1** and **T2**, so **notMemS0** can be used to latch the address.

notMemS1 goes low at the start of **T2** and the duration of its low period is programmable. It can, therefore, be used as RAS because RAS must go low at the beginning of **T2** and high at the beginning of **T5** to meet the precharge time.

notMemS2 has a programmable falling edge and goes high at the beginning of **T6**. It can, therefore, be

used as CAS. To allow sufficient data set up time during read cycles, and sufficient CAS/RAS lead time, **notMemS2** must fall at the beginning of **T3**.

We require one further signal, usually called Amux, which is used to switch between the row and column addresses supplied to the DRAM. Normally, as in the simple example, **notMemS2** would be used for this and **notMemS3** for CAS, leaving **notMemS4** for wait state generation but, in this case, we can make use of one of the features of the AAA280x series DRAMs: that of short row address hold time (tRL1AX), which is only 2 nsec. This allows the RAS strobe delayed by 2nsec or more to be used as Amux.

The preliminary circuit and timing are shown in figures 1.8 and 1.9.
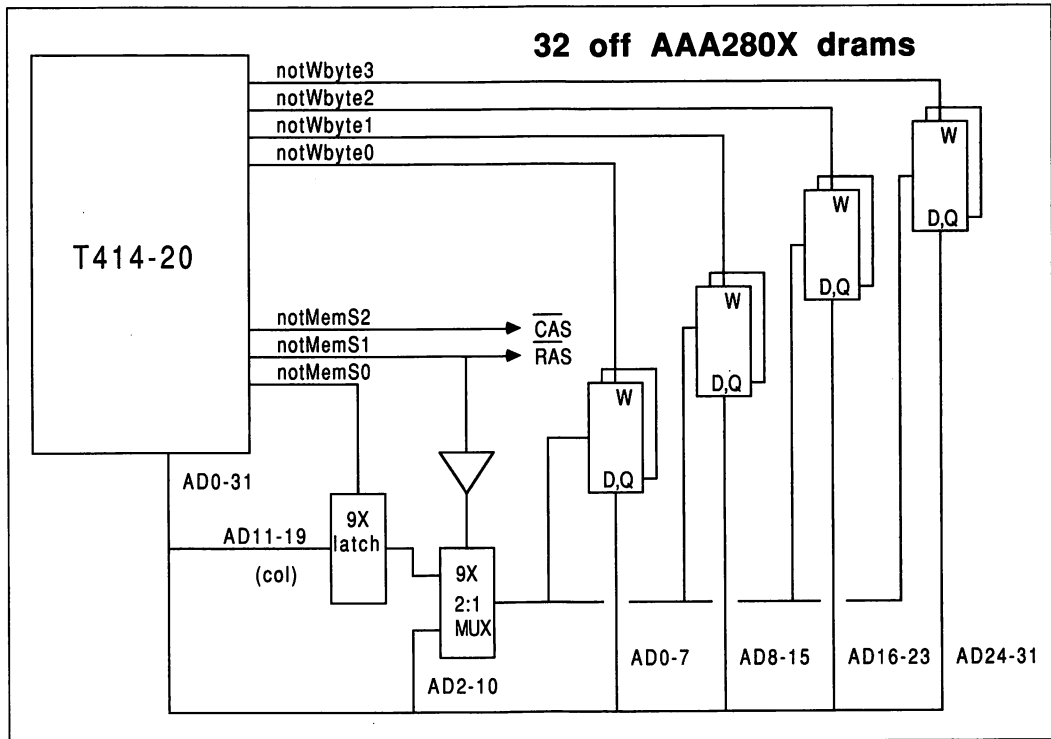
Figure 1.8

1.3.4   Address decoding

The RAM must occupy the bottom of the address space so that it appears to be a continuation of the transputer's internal RAM. The ROM must occupy the top of the address space, so that the transputer can boot from ROM. We can, therefore, use **A31** to select between RAM and ROM. **A2–A19** will be used to address the DRAMs so we should use **A20** to select between banks. We can also use **A20** to select between ROM and I/O. This gives a very simple decoding scheme:

A31	A20	
1	0	RAM bank 0
1	1	RAM bank 1
0	0	I/O space
0	1	ROM
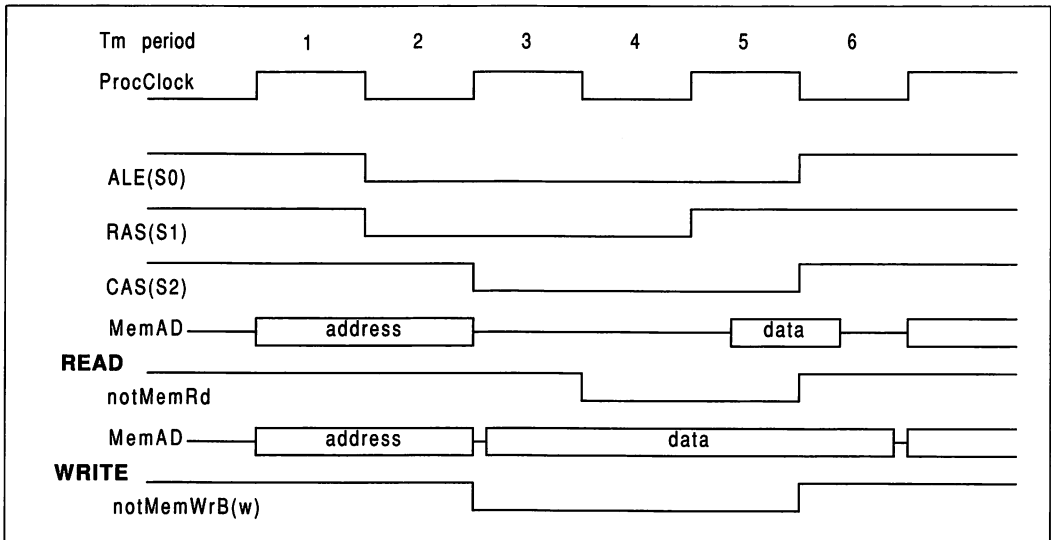
Figure 1.9

For most DRAMs: any RAS sequence will refresh an entire row of 1024 bits, reading or writing of data is initiated by CAS. Therefore, address decoding need only be applied to CAS; RAS can be enabled to both banks of RAM at all times. Thus, reading or writing one RAM bank will cause the other to be refreshed and accesses to ROM or I/O will refresh both banks.

Note that during a refresh cycle, **AD31** is low so that the CAS signals to both banks are disabled. Figure 1.10 shows the address decoding.
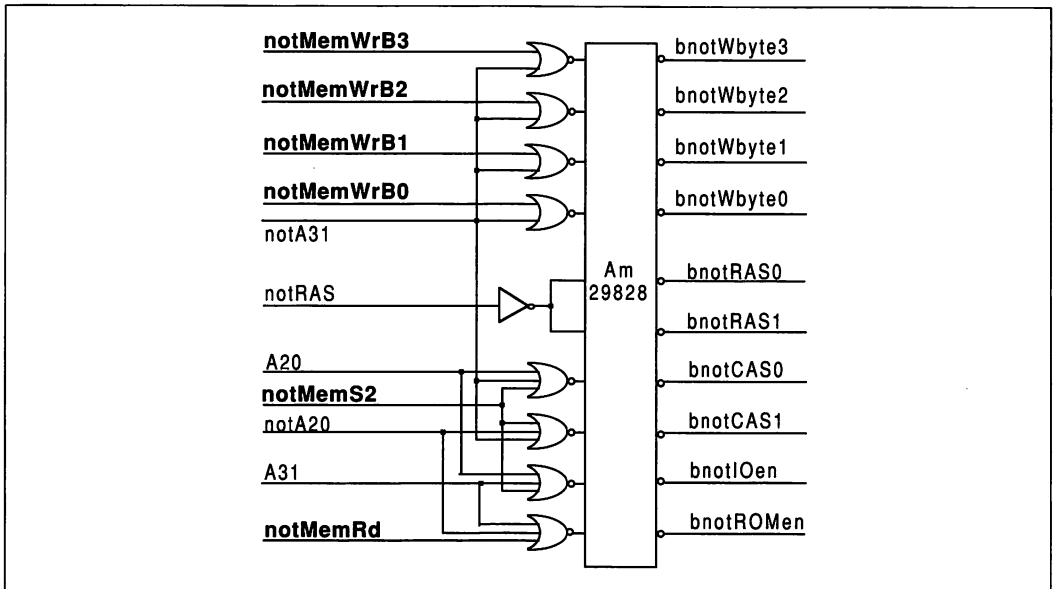
Figure 1.10

1.3.5   Loading considerations

The notRAS and notCAS signals will need to be buffered because each is required to drive 32 DRAMs, giving a total load capacitance on each line of:

$$32 \times 6 = 192 \text{ pF}$$

The four **notMemWrB** strobes will also require buffering as, for a 2 Mbyte memory, they must each drive 16 DRAMs giving a total capacitive load on each line of:

$$16 \times 6 = 96 \text{ pF}$$

The maximum load specified by INMOS is 50 pF.

Neither of these figures allows for layout capacitance so the actual load will be somewhat more.

We will choose to gate the **notMemWrB** strobes with some address decoding, prior to buffering them, so that they are not enabled to the DRAM when writing to peripherals.

1.3.6   Address latching and multiplexing

The address decoding requires that latched addresses should be valid as early as possible, and the most effective way to do this is with transparent latches. This way, the addresses will be stable before they are latched by **notMemS0**, so that the first stages of the decoding will already have settled. The complement of some of the address lines are also required by the decoding. These are provided by inverting the latched addresses.

The address multiplexing can be done by using an address latch with tri-state outputs and a tri-state buffer. The delayed RAS signal is used to switch between the buffer (row address) and latch (column address). Figure 1.11 shows the address latching and multiplexing circuit.
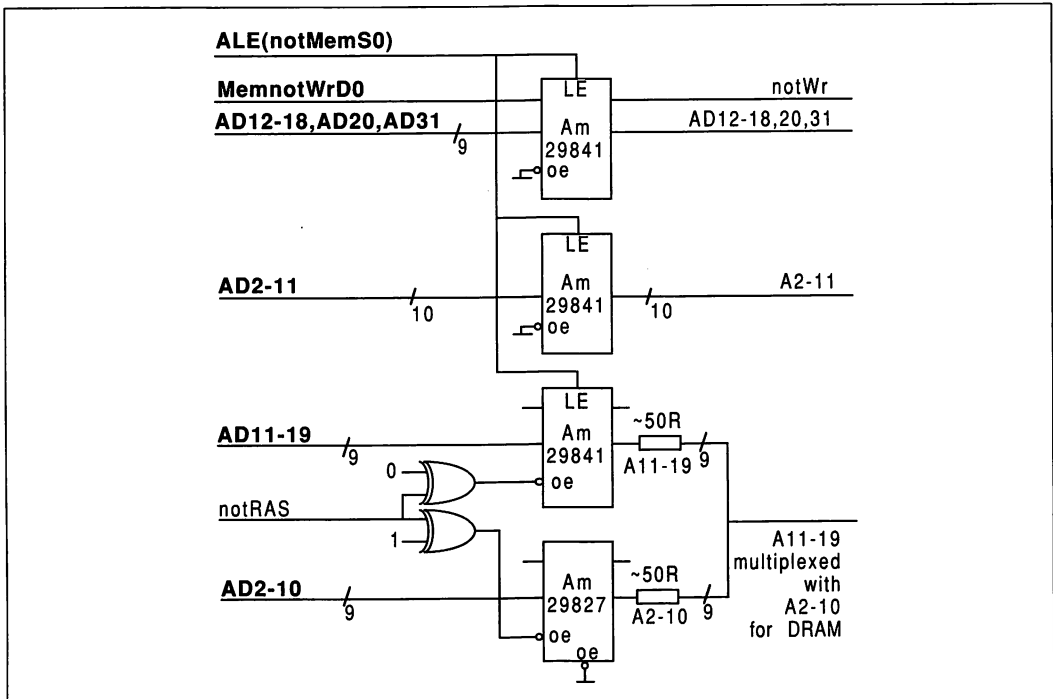
Figure 1.11

1.3.7   Evaluate DRAM timing

Since this is the most critical timing, and the one most subject to amendment, it should now be checked. This requires the drawing of a more detailed timing diagram than figure 1.9. The logic that has still to be added will not affect the timing.

The following steps then need to be followed to investigate the timing properly:

1 Add the skew of any signal change. From the T414 data sheet section on memory interface AC characteristics, this is, typically, −3/+4 nsec.

2 Add the propagation delays through any external logic, including any latches or buffers.

3 Check that all of the times on the data sheet for the DRAM devices in use are within specification.

4 If any parameter is outside the specification, try to meet it by altering the external logic or, if this is unsuccessful, insert extra **Tstates**.

The following table will be useful in determining propagation delays:

Device	Type	low-high in nsec	high-low in nsec
74F00	Quad 2i/p NAND	6.0	5.3
74F02	Quad 2i/p NOR	6.5	5.3
74F08	Quad 2i/p AND	6.6	6.3
74F27	Triple 3i/p NOR	6.0	5.3
74F32	Quad 2i/p OR	6.6	6.3
AM29828	10x inv. buffer	7.5 (14*)	7.5 (14*)

All 0–70 degrees C, worst case, load 50pF, *load 300pF

The emerging family of FACT HCMOS logic has superior characteristics to the FAST devices listed above, and is preferable where available. One of its main attributes is the symmetrical propagation delays which make it particularly suitable for buffering transputer links.

For most other logic, note that inverting logic generally has marginally lower propagation delays; thus if a gate has to be buffered, an extra 1–2 nsec can be gained by using say a NOR + inverting buffer over an OR + non-inverting buffer.

An examination of the resulting diagram, figure 1.12, shows one possible problem immediately: the write strobe may not go high until after the data bus has gone tri-state, causing data corruption on write with some RAMs. This is not a problem with page mode DRAMs which latch write data on the falling edge of CAS or Write, whichever is the later.

However, this potential problem can be completely removed by substituting a 74F32 for the 74F02 and removing the high-current buffer to reduce the propagation delay for the write strobes. The 74F32 can drive up to 180pF and the loading calculated in section 1.3.5, with an allowance for layout capacitance, is less than this. It is possible to use two 74F32s for each of the write strobes, one for each DRAM bank, to give lower propagation delays. This now provides the timing shown in figure 1.13.

The final selection of DRAM device can now be made. In this circuit RAS is used to switch the multiplexer and, since RAS goes high before CAS, the column address supplied to the RAM will change before the end of the CAS access cycle. Therefore, we must use a page mode DRAM (e.g. AAA2801 or uPD41256) which latches the column address on the falling edge of CAS, and is unaffected by subsequent changes.
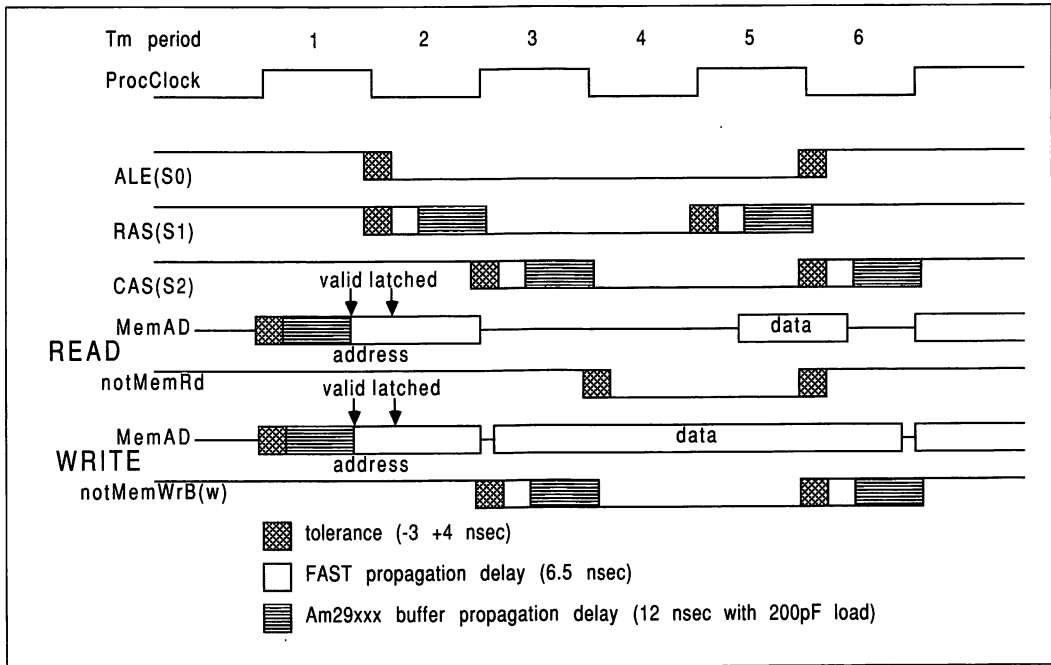
Figure 1.12
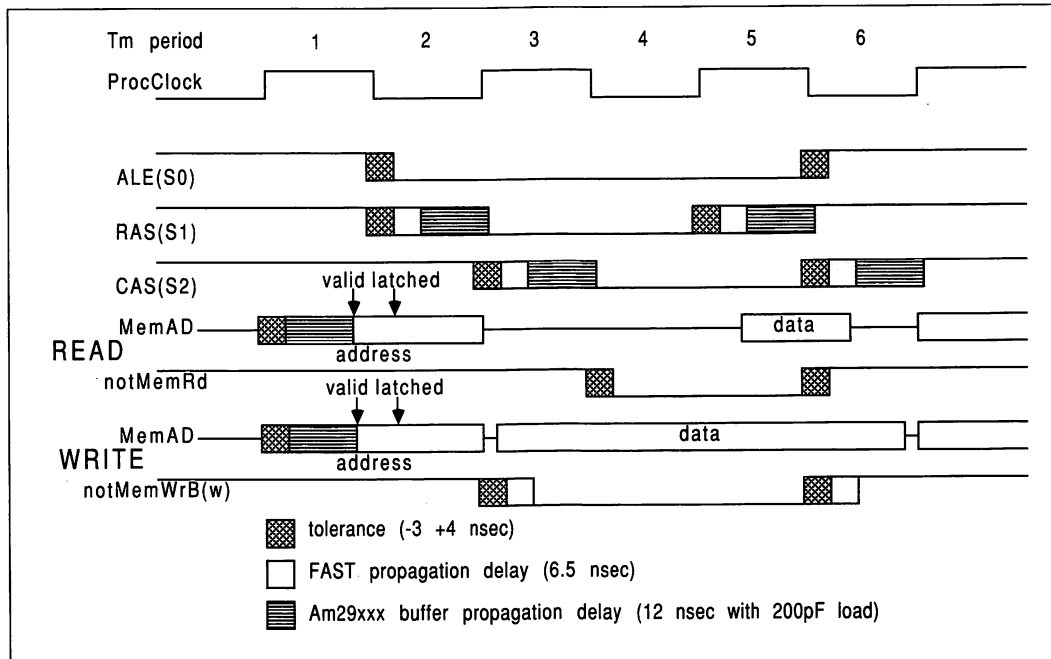
Figure 1.13

1.3.8   Choose write mode

Most DRAMs can perform two types of write cycle: early and late write. An early write cycle occurs when notWE is taken low before notCAS. Thus, the output buffers are turned off before CAS and the output pins remain tristate throughout a write cycle. A late write cycle occurs when notCAS is taken low before notWE. Thus, the beginning of a late write cycle appears to the DRAM to be a read cycle and read data is gated onto the output pins; this would be used in complex memory systems for read – modify – write cycles.

Early write cycles allow the DRAM's data input and data output pins to be commoned and connected directly to the AD bus. Late write cycles require the data output pins to be connected to the AD bus through tristate buffers enabled by **notMemRd**; otherwise the transputer AD pins and DRAM data output pins may collide in write cycles.

In this application, there is no requirement for late write cycles and the circuit will be simpler if we can achieve early write. This may be difficult because, to achieve sufficient read data set up time and RAS/CAS lead time, the falling edge of CAS (**notMemS2**) has been pulled forward to the beginning of **T3**. Hence, if the memory interface is configured for early write, the **notMemWrB** strobes fall coincident with **notMemS2**; i.e. coincident with CAS.

However, the heavier buffering on **notMemS2** means that notWE will become valid before notCAS and, because the early write set-up time (tWL1CL1) for the AAA280x series is only 0nsec, the DRAMs will experience early write.

Thus, the DRAM's data input and output pins can both be connected directly to the AD bus.

The DRAM circuit has now been worked through and it remains only to choose the refresh interval and add EPROM and peripherals.

1.3.9   Choose refresh interval

Most 256k DRAMs are organised as 256 rows of 1024 bits each row of which must be refreshed within 4 msec if data is not to be lost.

The memory interface program gives the time taken for 256 refresh cycles based on the input clock frequency and the refresh interval. In this example, with a 5MHz input clock, the longest refresh interval of 72 clockin periods gives 3.69 msec for 256 cycles, within the maximum of 4 msec allowed for the DRAMs used.

1.3.10   Timing for other memory and peripherals

notMemRd is used to generate the EPROM chip select because, in the default memory configuration used to read the memory configuration word from ROM after reset, it is the only available strobe. **notMemS2** is used to generate the peripheral chip select because, since it goes high at the beginning of **T6**, its low period is stretched by wait states; whereas the low period of **notMemS1** is fixed. The address decoding shown provides one wordwide ROM/EPROM space and one I/O space.

The timing for a common medium speed EPROM is typically:

t access	200 nsec	access time
t ce	200 nsec	chip enable time
t oe	75 nsec	output enable time
t df	60 nsec	output turn off(to bus float)

Access, chip enable and output enable times can all be met by the use of wait states with the timing already derived. However, Tdf is another problem. Referring to figure 1.10 and 1.13, it can be seen that peripheral and ROM/EPROM enable timing will be the same as CAS except for the wait states inserted between **T4** and **T5**. Thus Tdf is restricted to a limit of 0 nsec if the bus is to be tri-state by the start of **T1**, when the addresses are placed on it. Using **notMemS2** directly, rather than buffered, which is possible if the loading is not exceeded, will give 12–15 nsec available, but this is considerably less than that required. The Tdf of typical peripheral devices, such as the SCN2681A DUART (DUal Asynchronous Receiver / Transmitter) is up

to 100 nsec, compounding the problem.

There are two basic routes to a solution; the first is to rearrange the timing, but this will slow down the DRAM cycles as well, thus defeating the object of this design. The second is to use external buffers on the data lines connected to ROM and peripherals. The delay through these buffers must be taken into consideration when determining the number of wait states required.

If F245 buffers are used, these should be enabled by **notMemRd** or **notMemWrB** during ROM or peripheral access cycles. These strobes must be used because they are the only ones available in the default memory configuration after reset. The direction can be selected by the latched **MemnotWrD0** signal. This is low during **T1** and **T2** of a write cycle and can, therefore, be latched in the same way as the address.

Thus, all that remains to be designed is the gating logic for the wait state generator. This must gate **notMemS4** to **MemWait** during ROM access cycles, and **notMemS3** to **MemWait** during peripheral access cycles; during RAM access cycles and refresh **MemWait** must be held low. **notMemS4** is used as the wait state generator for ROM accesses because it alone will generate a suitable length of wait state in the default memory configuration after reset. The NAND gate is included in the address decoding for ROM and peripherals to ensure that wait states are not inserted in refresh cycles; when **A20**=1 and **A31**=0.

Figure 1.14 gives the full detail of the circuit, and although this represents a complex design by transputer standards, it is still very simple when compared to the support logic required for other processors in a similar system. Memory configuration data is taken from EPROM, on data line 0. Figure t09:FAST2 shows the final timing, without the wait states for EPROM and I/O.
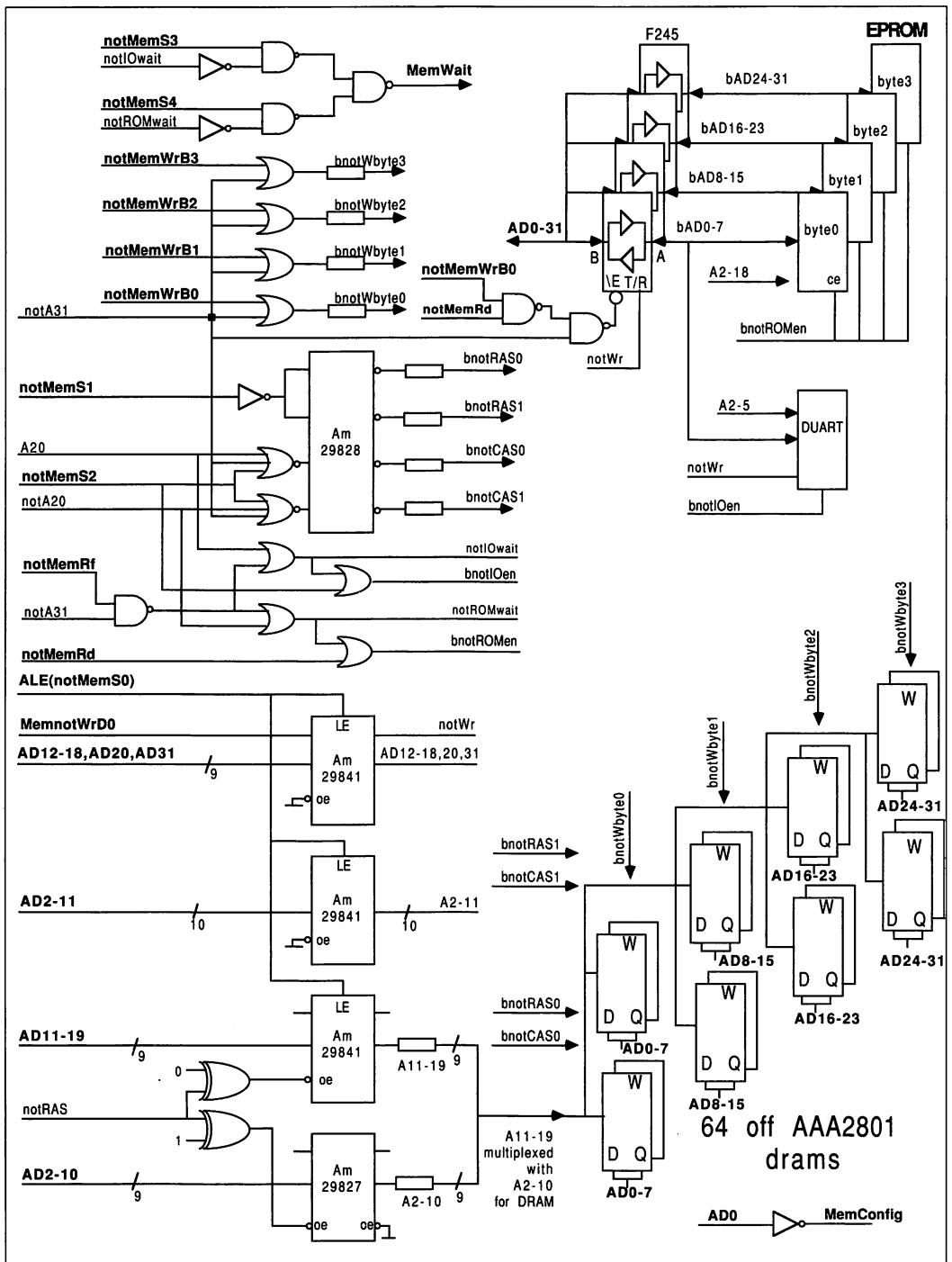
Figure 1.14

1.3.11   Summary of design steps

As each application is different, it is hard to generalise, but figure 1.15 is a flow chart showing the major steps. In all systems, it is necessary to start with the RAM timing, as that is the most critical area, and will have the greatest impact on system performance. In many designs, RAM is the only memory.

```
          ┌──────────────────────┐
          │   Select transputer  │
          └──────────┬───────────┘
                     │
          ┌──────────▼───────────┐
          │   If using DRAM,     │◄──────────────┐
          │ select RAS duty cycle│               │
          └──────────┬───────────┘               │
                     │                            │
          ┌──────────▼───────────┐               │
          │ Select no. of Tm periods             │
          │   for memory cycle   │               │
          └──────────┬───────────┘               │
                     │                            │
          ┌──────────▼───────────┐               │
          │    Allocate DRAM     │               │
          │   control strobes    │               │
          └──────────┬───────────┘               │
                     │                            │
          ┌──────────▼───────────┐               │
          │    Check loading,    │               │
          │  buffer if necessary │               │
          └──────────┬───────────┘               │
                     │            ┌───────────────┴──┐
          ┌──────────▼───────────┐│   Make change    │
          │    Plot out timing   ││  and reiterate   │
          └──────────┬───────────┘└──────────────────┘
                     │                       ▲
                  ◄DRAM►                      │
                 Is DRAM ──────────────────►──┘
               timing within
                   limits?
                     │
          ┌──────────▼───────────┐
          │     Add in other     │
          │    memory types      │
          └──────────┬───────────┘
                     │
          ┌──────────▼───────────┐
          │    Plot out timing   │
          └──────────┬───────────┘
                     │
                  Is it all ─────────────────►
               within limits?
                     │
          ┌──────────▼───────────┐
          │ Build circuit and test│
          └──────────────────────┘
```
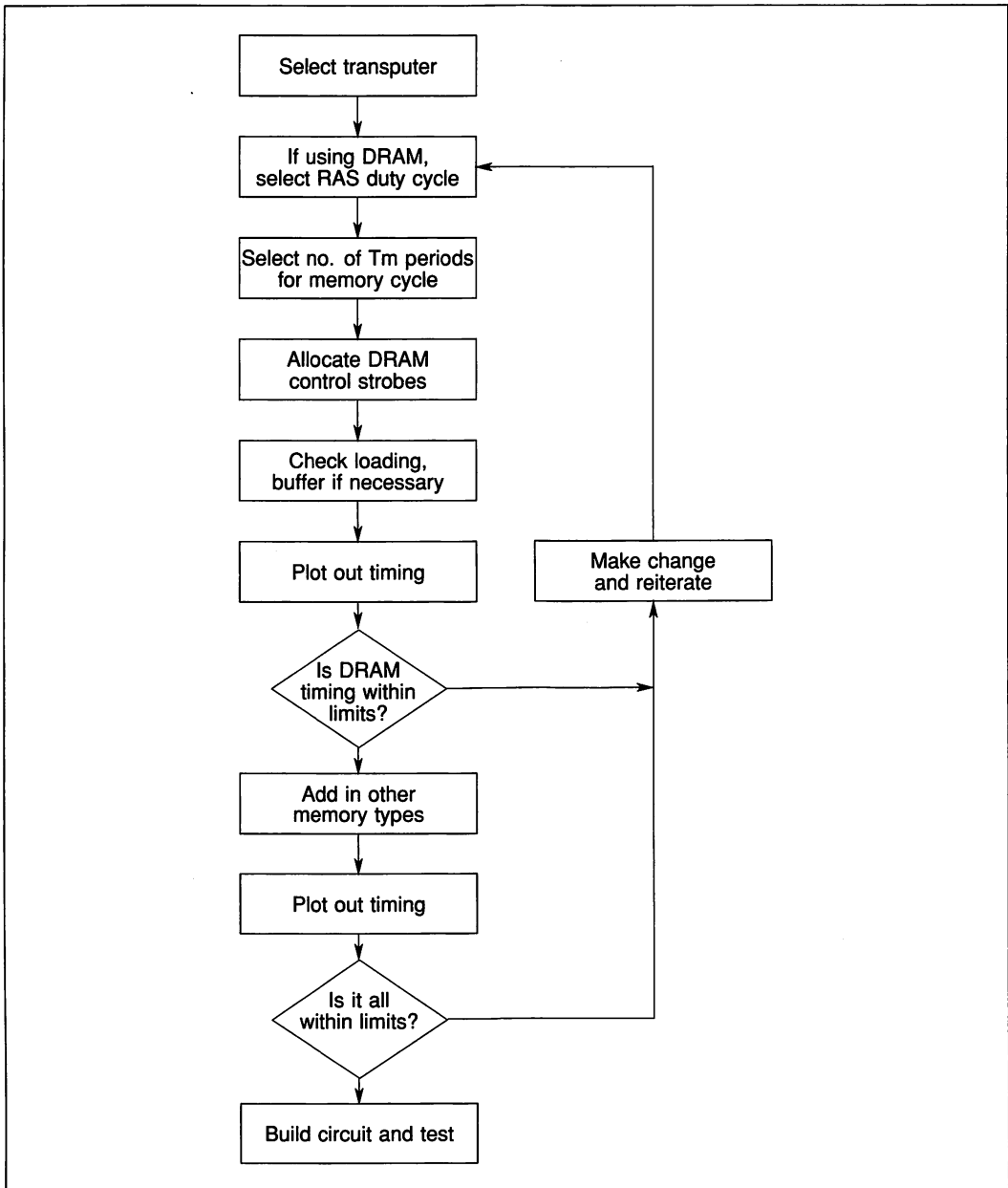
Figure 1.15

1.4   Further examples

1.4.1   Minimum component, 256Kbyte memory

The example in figure 1.16 is taken from the Inmos B003 board. On this board, the 256k byte memory is made up of eight 64k x 4 DRAMs (e.g. NEC uPD41464).
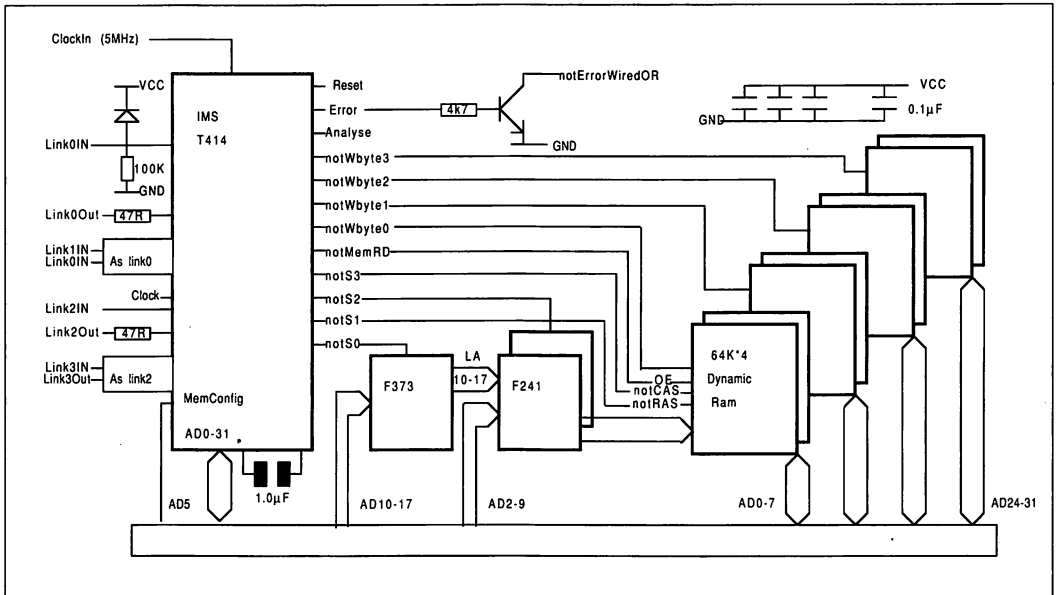
Figure 1.16

NotMemS0 is used to latch address bits 10–17 into a 74F373 and two 74F241s are used as an addresss multiplexer. **NotMemS1** is used as notRAS, **notMemS2** is used as the select on the multiplexer and **not-MemS3** as notCAS. Each **notMemWB** strobe goes to a pair of 64k x 4 DRAMs and **notMemRD** goes to all. Thus, the 256k bytes is organised as 64k words of 32 bits. The internal memory configuration selected by connecting **MemAD5** to the **MemConfig** input is used; figure 1.17 shows the timing in terms of **Tm** periods, so the transputer clock speed has to be taken into account before actual timings can be added to the diagram.

It is possible to reduce the component count still further by using devices such as the 74F604/6. This is a 16 bit latch to 8 bit multiplexed output, one version being faster and the other glitch free. The only drawback of this device is that the latches are rising edge triggered and, therefore, an inverter is needed in **notMemS0**. Again, care must be taken to ensure that the loadings on RAS and CAS are not exceeded. Figure 1.18 outlines this circuit.

In simple systems, the use of transistors or power MOSFETs can keep the required board area down. Power MOSFETs such as the Motorola MPF910 make useful drivers, as they come in a TO92 package, can handle peak currents in the range 1–2A, and have turn on/turn off times of 4 nsec; thus they can charge or discharge a large capacitance very quickly. The careful use of discretes such as these can allow better board layout and allows more control of the heavy currents that flow during switching.
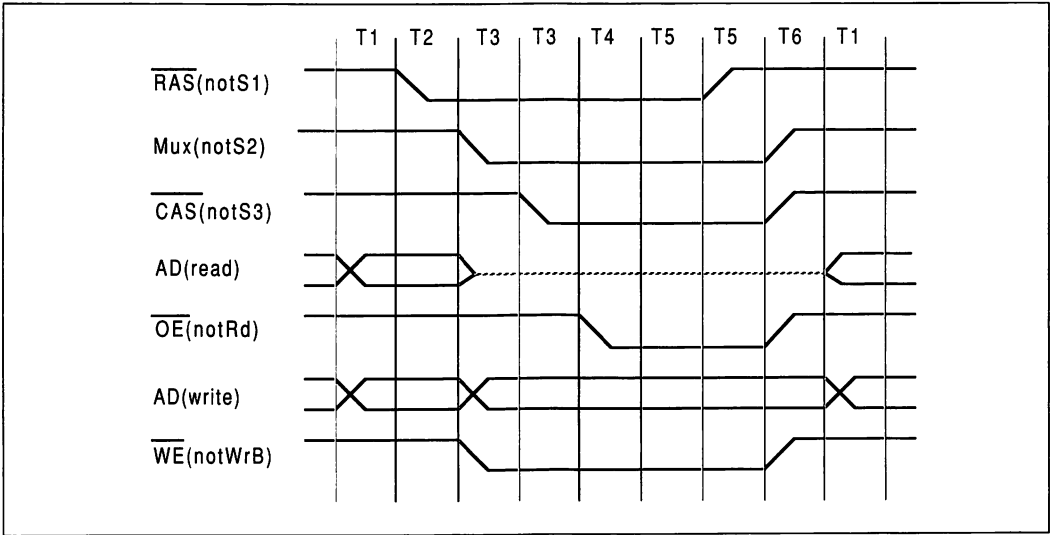
Figure 1.17
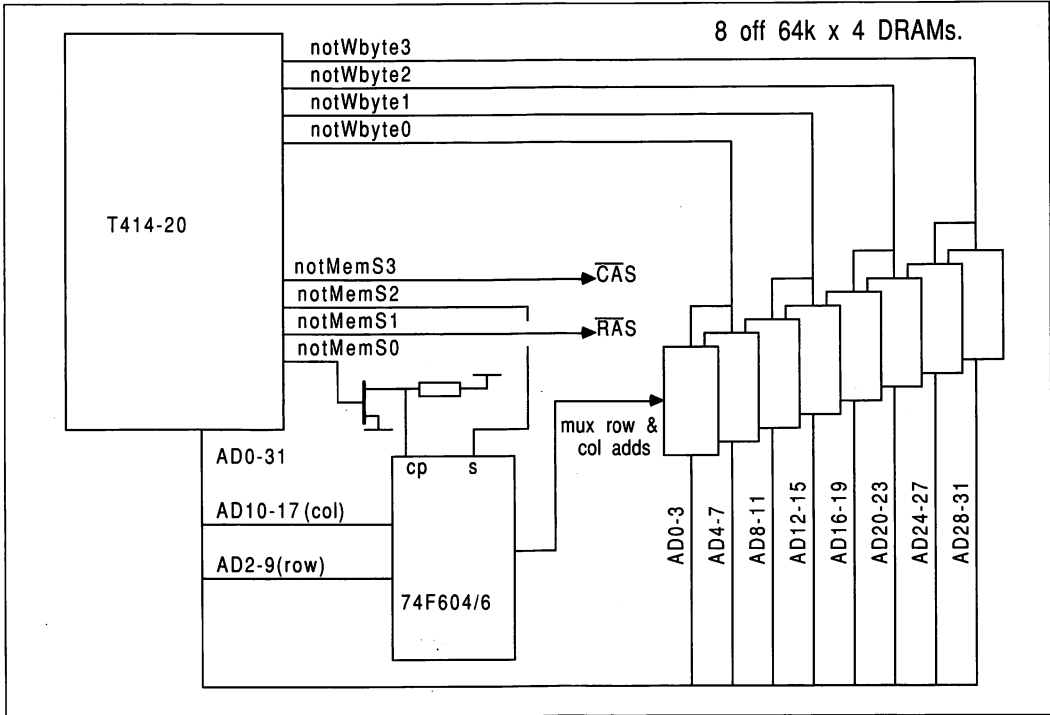
Figure 1.18

1.4.2   DRAM only: 1 Mbyte

This has been outlined during the main worked example, but is detailed here in its minimum form. The row and column address multiplexer is made from a tri-state latch and buffer. As this is a RAM only system, and there is only one bank of RAM, no address decoding is required and it is not necessary to detect refresh cycles. Instead, refresh cycles can be allowed to appear to the RAM as normal read cycles and they will still have the desired effect.
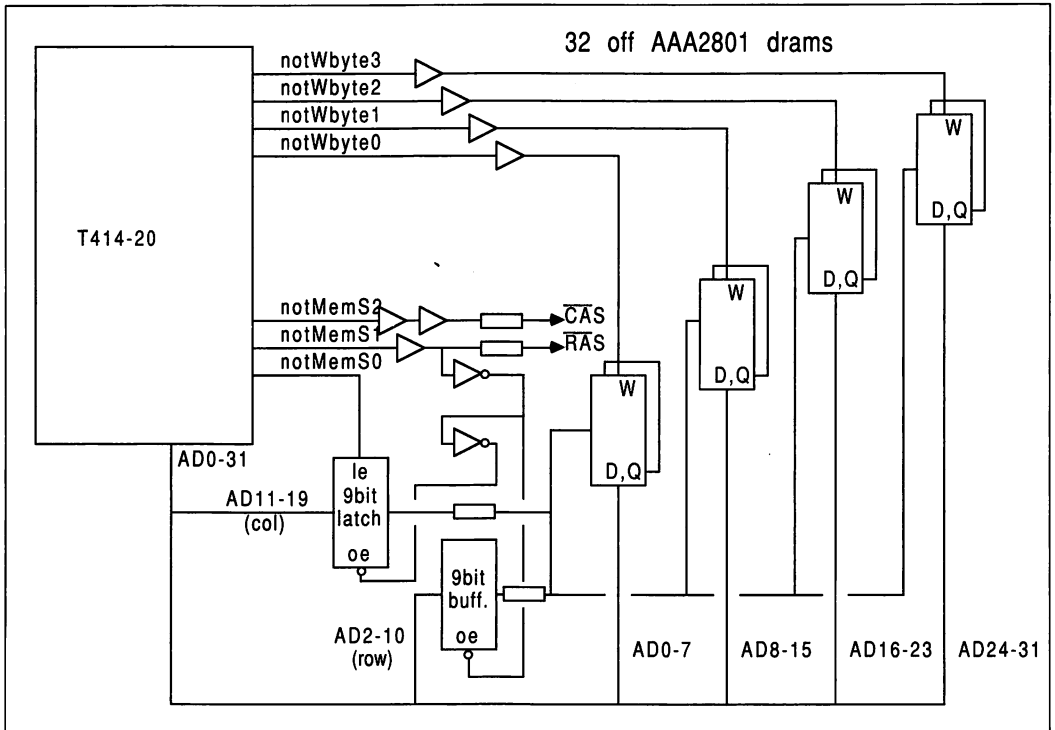
Figure 1.19

In the circuit shown in figure 1.19, RAS delayed by a gate is used as Amux. This allows CAS to go low one **Tm** period after RAS goes low, giving a longer access time and, hence, the shortest possible memory interface cycle time; 3 cycles of **ProcClockOut**. With longer cycle times, it is possible to use **notMemS2** for Amux and **notMemS3** for CAS. Note that to ensure early write, CAS has been delayed with respect to the write strobes by an extra buffer.
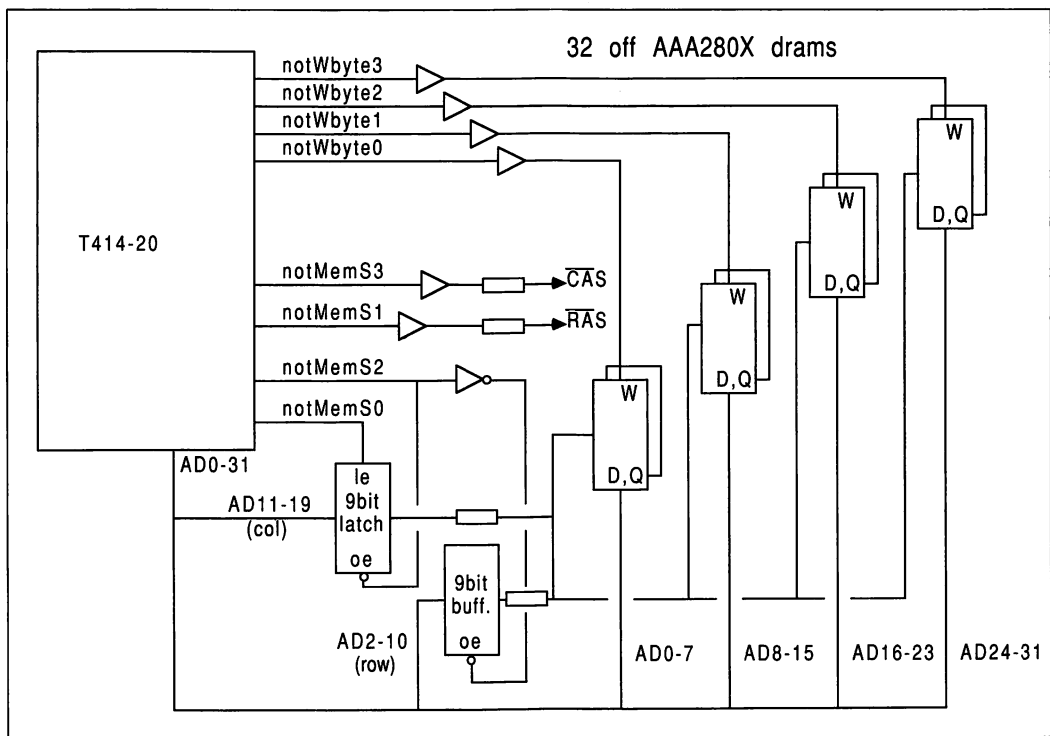
Figure 1.20

If very fast memory devices are available, it may be possible for CAS to fall at the beginning of **T4** and still achieve a memory cycle time of 3 cycles of **ProcClockOut**. In that case, Amux can be generated by another strobe, as there will then be two **Tm** periods between RAS and CAS. This is shown by the circuit diagram, figure 1.20, and the timing diagram, figure 1.21.

The important parameters to consider here are the CAS to RAS lead time, the time from CAS going low to RAS going high, and the CAS access time. The CAS to RAS lead time is a minimum of 15 nsec for the AAA2800-60, adding the transputer tolerances to the strobe edges allows about 18 nsec; if a greater margin is required, inserting an extra buffer in RAS will provide it. For the AAA2800-60, CAS access time is 11nsec maximum, so the buffer delay on CAS must be minimised to give sufficient access time. Thus, it may just be possible to do this with AAA2800-60 DRAMs.

The circuit in figure 1.20 could be extended to 4 Mbytes by substituting 1 Mbit DRAMs for the 256k DRAMs but, with current memory speeds, 4 cycles would be needed for the memory interface.

Tm period   1   2   3   4   5   6

ProcClock

ALE(S0)

RAS(S1)

Amux(S2)

CAS(S3)   valid latched

READ   MemAD   data

        notMemRd   valid latched

WRITE   MemAD   data
                          address

        notMemWrB(w)

        ▨ tolerance (-2 +5 nsec)

        ☐ FAST propagation delay (6.5 nsec)

        ▤ Am29xxx buffer propagation delay (12 nsec with 200pF load)
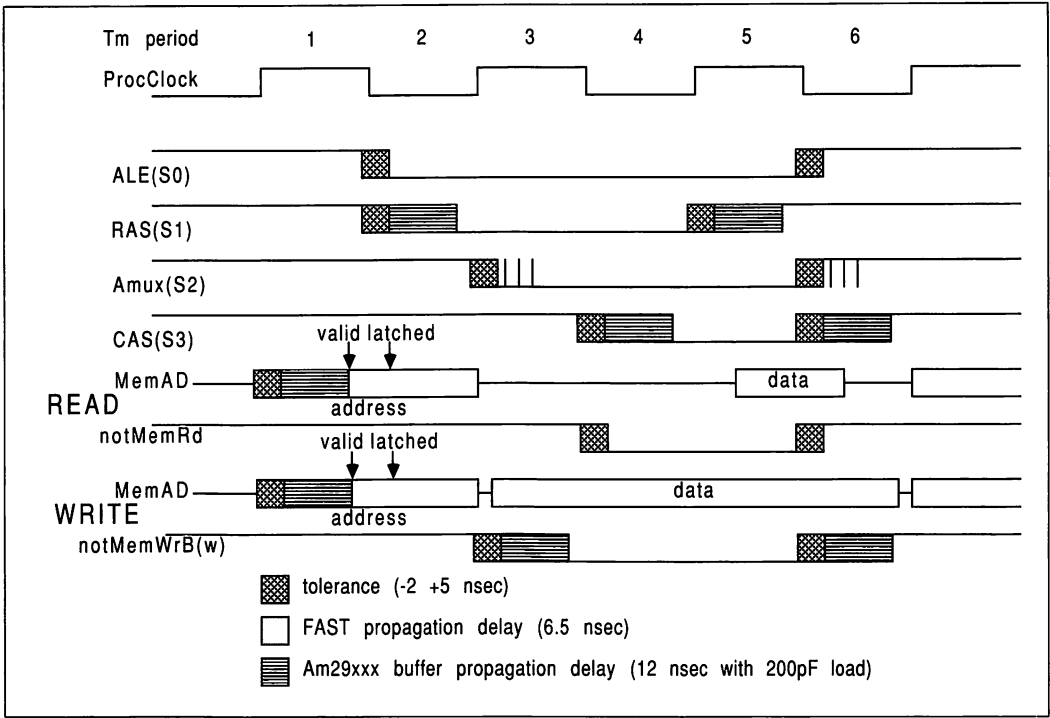
Figure 1.21

1.4.3   Fast static memories

Other than the problem of meeting access time, the only critical timing is the chip disable to output inactive time. For either the IMS T414 or the IMS T800 to acheive the fastest possible memory cycle time this must be less than one **Tm**. Static RAMs with common data IO pins generally have faster turn-off times than those with separate IO. The following table gives the most important times for the IMS 1620 (16k x4) and the IMS 1820 (64k x4).

Memory	1620-45	1620-55	1620-70	1820-25	1820-35	1820-45[1]	
Access time	45	55	70	25	35	45	nsec
Write pulse width	40	50	60	20	30	40	nsec
Chip disable to output inactive	20	25	25	15	15	20	nsec

It is possible to operate static RAMs in two modes: asynchronous, where the device is continuously enabled, or synchronous, where the address inputs are only allowed to change when the device is deselected. Synchronous operation is preferred because it achieves lower error rates than asynchronous operation. Synchronous operation is very easy to implement with the IMS T414 or IMS T800, by using one of the programmable strobes as chip enable.

[1]Product under development. Contact INMOS for availability.
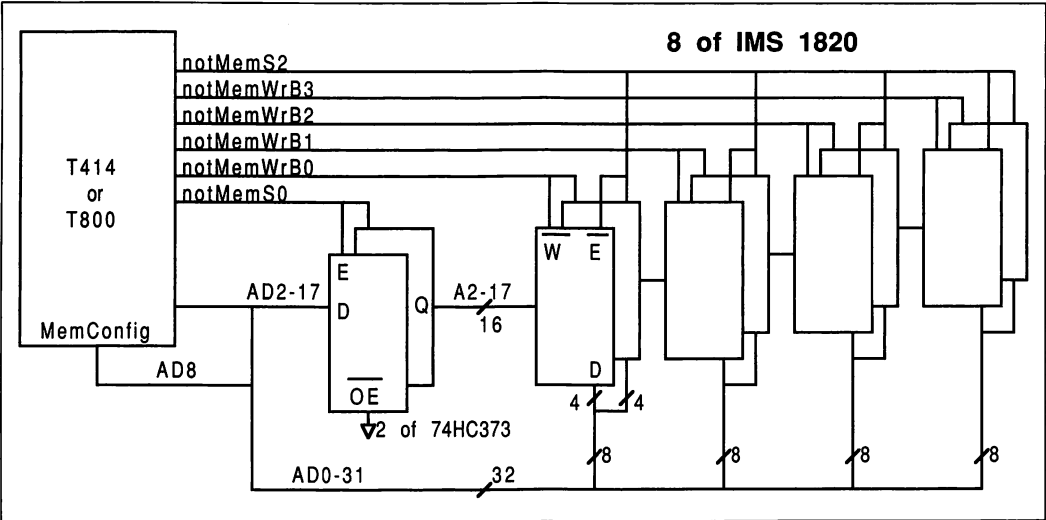
Figure 1.22

The memory configuration used is very simple; figure 1.23. One with early write is preferred as this allows slower memories to be used. For example, with a T414-20 or T800-20, the IMS 1620-45 (for total 64Kbytes) or IMS 1820-45 (for total 256Kbytes) can be used. For the IMS T800-30, the IMS 1820-35 should be used.

Expansion of the system illustrated above is easy until the bus loading becomes too great or until address decoding is needed. Any address decoding must impose a minimal delay on chip enable as any delay reduces the available access time and also the time available for disabling the RAM output buffers. If the delay through the address decoding is too great, a slower memory cycle can be used.

Alternatively, if data bus buffers are used to reduce the bus loading, these will turn off faster than the RAM output buffers and it may not be necessary to use a slower cycle.
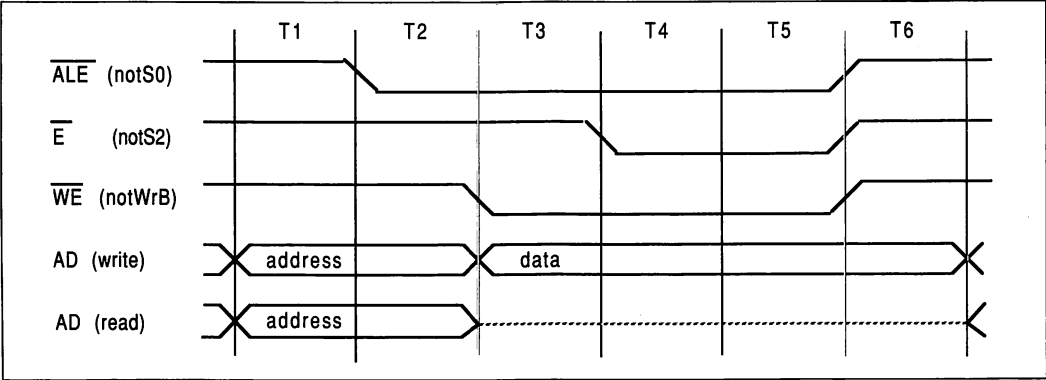
Figure 1.23

1.5   Debugging memory systems

1.5.1   Peeking and poking

Transputers can be booted from ROM (**BootFromROM** to Vcc) or from link (**BootFromROM** to ground). When booting from link, a header byte is expected, if it is in the range 2–255 it should be followed by that number of bytes. These will be placed in memory starting at **MemStart** ($80000048) and execution will then be transferred to this address. The code executes at low priority and its work space is located immediately above itself. Usually, this code will be a loader, to load the user's program into this transputer and any others, if it is part of a network.

If the header byte is 0, a 'poke' operation will take place. The 0 byte should be followed by a 4 byte address (AAAA) and 4 bytes of data (DDDD) to be placed at that address:

```
input:   header=0, then A A A A D D D D
```

If the header byte is 1, a 'peek' operation will take place. The 1 byte should be followed by a 4 byte address (AAAA). The transputer will then output, on the same link, 4 bytes of data (DDDD) read from that address:

```
input:   header=1, then A A A A
output:                  D D D D
```

After both the peek and poke operations, the transputer reverts to awaiting a new header (which could initiate another peek or poke).

Thus, if the user has another transputer, such as the one in the development system, it is possible to test the hardware by poking to the transputer under test to place data in the internal or external memory, and then peeking to read the data back and compare it. The same method can be used to test, say, a UART. These peek and poke operations allow simple test programs to be written in occam and run on the development system, considerably simplifying the design engineer's job. For temperature range testing, the system under test can be put in an environmental chamber with development system outside; all that is needed to connect them is a reset cable and a 4 wire link cable. In a mixed memory system, the engineer can now determine whether it is the memory or the DUART that is marginal, something that previously was difficult to do.

1.5.2   Investigation of memory timing

There may be occasions where a designer wishes to compare different memory interface configurations, and rather than programming an EPROM or a PAL in order to alter a parameter each time, software configuration for the memory interface would be useful. In figure 1.24, a basic scheme is outlined for this. It assumes that a known working transputer board is available, such as one that is part of the development system. This is used to 'poke' the required parameters into the RAM, which need only be one bit wide, as previously described for memory debugging; the memory configuration used is the internal configuration associated with ADx. Poking anything to a location of $8xxxxxxx will then generate a reset and cause the new memory configuration to be read from RAM on the line ADx. The memory debugging technique can then be used to test the system. Pressing the reset switch will generate a new reset and select the internal configuration again. Thus, once a software configuration has been selected, it cannot be altered by any program that may be run.
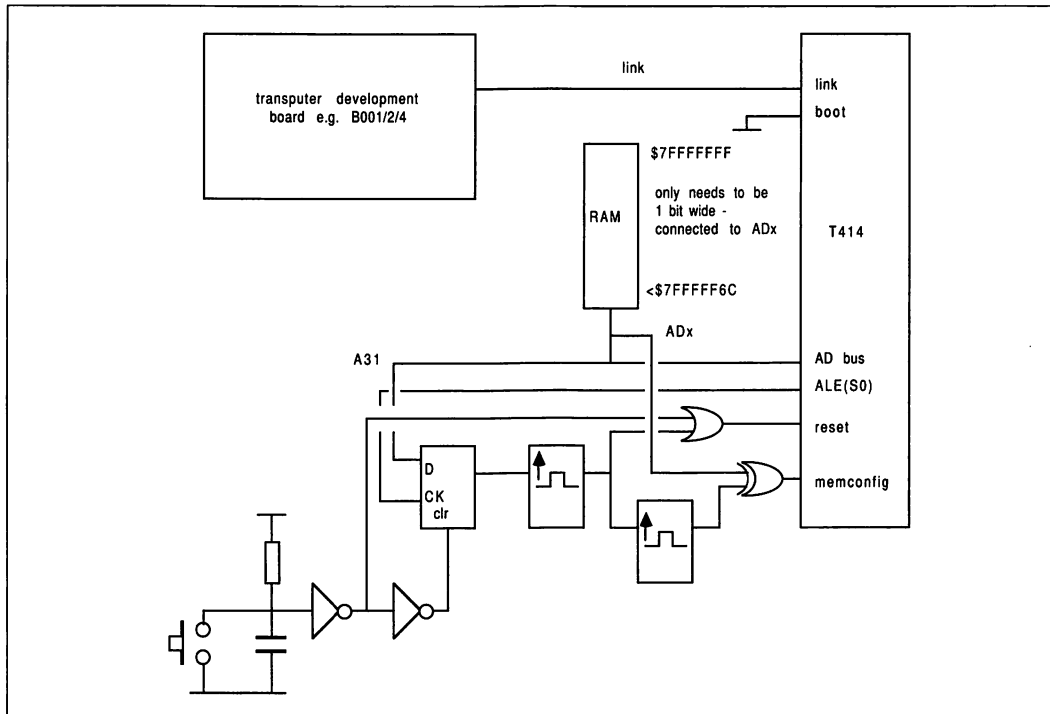
Figure 1.24

1.6   Summary

Whilst this document has not covered the memory interface of the T414 transputer exhaustively, it has shown the main features and how complex systems can be built with the minimum of effort. The reduced amount of logic required means fewer problems with propagation delays and race and hence faster memory cycles and shorter design cycles.

2 Connecting INMOS links

2.1 Introduction

The INMOS link is fundamental to the concept of the transputer and of occam [1, 2]. A link is the hardware implementation of an occam channel, each bidirectional link providing a pair of occam channels, one in each direction. A link provides serial data communication between two transputer family devices at speeds up to 20Mbits/s.

A link between two transputers is implemented by connecting a link interface on one transputer product to a link interface on the other transputer product by two uni-directional signal lines. Each signal line carries data and control information.

Communication through a link involves a simple protocol. This provides the synchronised communication of occam. The use of a protocol providing for the transmission of an arbitrary sequence of bytes allows transputer products of different wordlength to be connected together.

Electrically, link signals are TTL compatible and as such are a simple means of communication over short distances (< 0.3 metre). Links are designed for local communication. However, it is possible to use them over longer distances although a little more consideration is needed to ensure reliable operation. This application note is intended to provide the kind of information needed to engineer reliable links over various distances and media.

The note describes the operation of the INMOS link protocol followed by a discussion of the adverse phenomena encountered in link transmissions and means by which they may be overcome. Finally, a 5Mbits/s fibre optic link is described.

2.2 Link operation

An INMOS link between two transputer products consists of two uni-directional signal lines connected to the link interface on each transputer family device, providing point-to-point serial communication, as shown in figure 2.1.
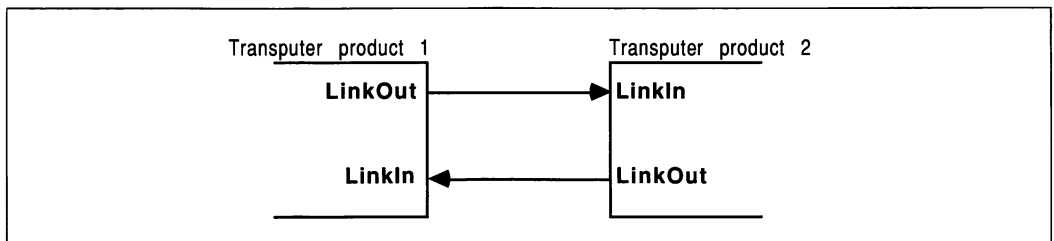
Figure 2.1 Link connection

Communication across a link involves a simple protocol (figure 2.2).

Each message is transmitted as a sequence of single byte communications, requiring only the presence of a single byte buffer in the receiving transputer to ensure that no information is lost.

Each byte is transmitted as a start bit then a one bit, followed by the eight data bits and a stop bit.

After transmitting a data byte , the sender waits until an acknowledge is received. This consists of a start bit followed by a zero bit. The acknowledge signifies both that a process was able to receive the acknowledged byte, and that the receiving link is able to receive another byte. Acknowledges may not be sent in advance. The receiving end starts with an empty buffer, ready to receive the first byte. The sending link reschedules the sending process only after the acknowledge for the final byte of the message has been received.

Data bytes and acknowledges may be multiplexed down each signal line during duplex communication. In
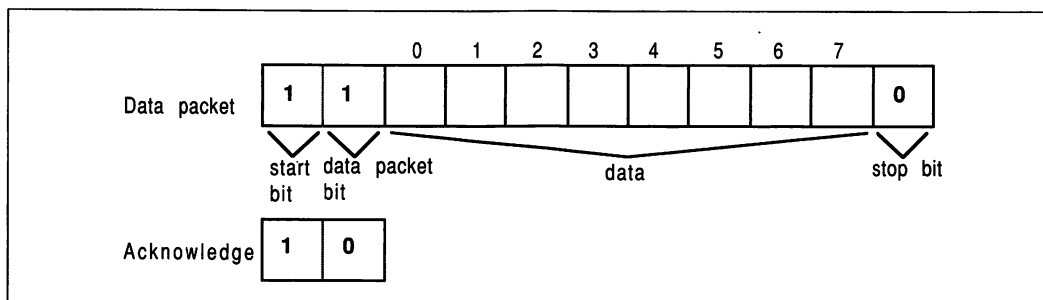
Figure 2.2 Link protocol

one implementation of the link (e.g. IMS T414) acknowledges are output on receipt of the full eleven bits of the data packet. The link implementation provided on the IMS T800 or IMS T222 allows overlapped acknowledges. In this implementation, the acknowledge may be sent immediately on receipt of the start bit *and* the 'data is to follow' bit, allowing continuous data transmission with no delays between data packets.

The quiescent state of a link output is logic '0', i.e. 0V.

2.3   Electrical considerations

Links may be connected very simply over short distances (<0.3 metre). No engineering is required other than a direct wire connection between LinkOut of one transputer and LinkIn of another. The connection may consist of tracks on a pcb or backplane, or a cable.

Over greater distances, certain parameters of the interconnection medium must be taken into account:

> Transmission line effects
>
> Noise and crosstalk
>
> Line attenuation
>
> Pulse dispersion
>
> Skew
>
> Propagation delay

A further consideration that applies to all link connections is protection of the link interface from electrostatic discharge.

This application note discusses these parameters as they apply to INMOS links. The communications medium commonly used at present by INMOS is twisted pair cable. The discussion of link parameters concentrates on this medium, but it could apply equally well to other transmission media, e.g. coaxial cable.

2.3.1   Transmission lines

INMOS links are designed to transmit serial data between transputer family devices at speeds up to 20Mbits/s.

The signals are TTL compatible and as such are suitable for transmitting data over short distances (up to 30cm) with no engineering except a simple wire connection.

At greater distances, the wire will exhibit transmission line effects which can cause undesirable *undershoot* or *ringing* in the received signal.

This section discusses why these effects occur and means by which they may be alleviated.

The transmission line
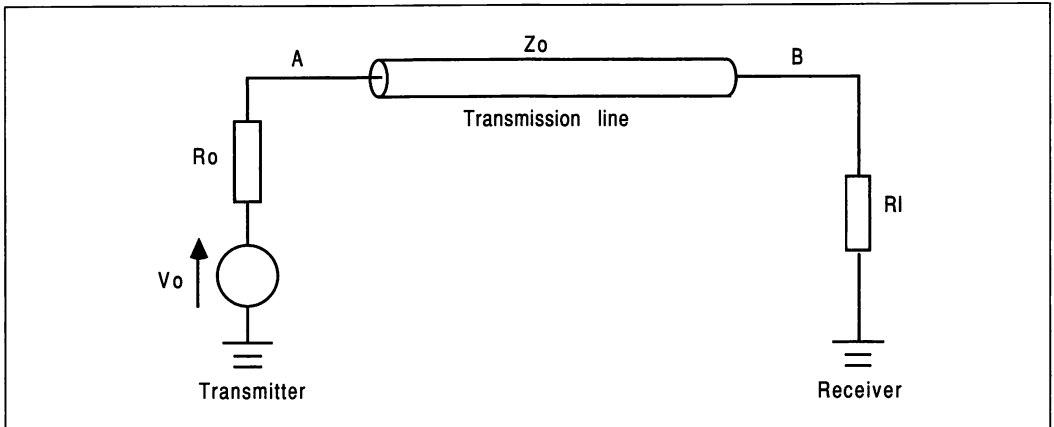
Figure 2.3 Typical transmission system

Figure 2.3 shows a typical transmission system. As the length of the transmission line is increased signals travelling through it are delayed. Transmission line effects take place when the propagation delay is significantly greater than 33% of the risetime of the transmitted digital signals, manifesting themselves as ringing and undershoot, as shown in figure 2.4.
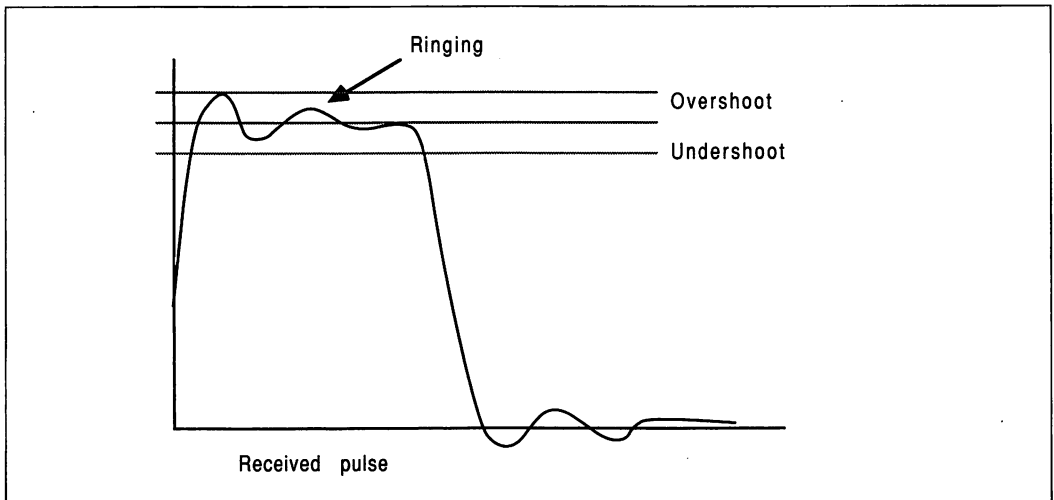
Figure 2.4 Transmission line effects

The 10–90% rise and fall times of the link outputs varies with capacitive loading, as shown in figure 2.5. [3]

As can be seen, the minimum rise time of 12ns corresponds to a capacitive loading of 20pF.
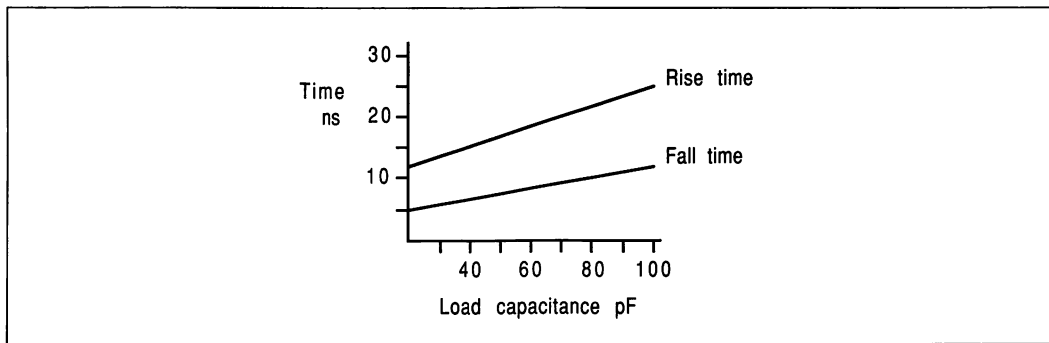
Figure 2.5 Typical link rise/fall times

Transmission line effects become significant when the length of the transmission line is one tenth of the wavelength of the highest frequency component in the transmitted signal. i.e.

$$l = 0.1\lambda$$
$$= \frac{0.1 v_p t_r (ns)}{350.10^6}$$

Thus, the effects begin when the delay down the line is

$$t_d = \frac{0.1\lambda}{v_p}$$
$$= \frac{0.1 t_r}{350.10^6}$$
$$= \frac{t_r}{3.5}$$

Where

l = length of the transmission line(m)
λ = wavelength(m)
v_p = propagation velocity of the signal through the line(m/s)
t_r = rise time of signal(ns)

Thus, for a rise time of 12ns, transmission line effects will occur when the delay down the line is greater than 3.4ns.

A typical value of v_p for twisted pair cable is 60% of the velocity of light. Thus, a propagation delay of 3.4ns is equivalent to a length of 60cm.

Figure 2.5 shows that the fall time is generally half the rise time for a given capacitive load. Thus, the frequency components in a falling edge will give rise to transmission line effects when the line length exceeds half that of the rise time minimum length. i.e. 30cm.

Transmission line effects

A transmission line has associated with it a *characteristic impedance*, Z_o. This is dependent on the inductance and capacitance per unit length and is given by

$$Z_o = \sqrt{\frac{L_l}{C_l}}$$

Where

L_l=inductance per metre
C_l=capacitance per metre

Consider a rectangular pulse sent along a transmission line. The rising edge of the pulse travels along the line arriving at the receiver after a propagation delay T_d, determined by the capacitance and inductance of the line, and causes a voltage drop across the load resistance R_l, giving rise to voltage v_b. Depending on the value of the load resistance, a *reflection* may occur which will travel back down the line to the transmitter. The amplitude of the reflected voltage depends on the *reflection coefficient*, given by

$$\rho = \frac{R_l - Z_o}{R_l + Z_o}$$

The amplitude of the reflection is given by ρ . v_b. Clearly, if $R_l = Z_o$, ρ is zero and no reflection takes place. In the worst case, if $R_l >> Z_o, \rho = 1$; if $R_l << Z_o, \rho = -1$. If a reflection occurs, the reflected pulse travels back down the line arriving at the transmitter after another propagation delay T_d. If the output impedance of the transmitter is not equal to Z_o, another reflection takes place which travels back to the receiver where a further reflection takes place, and so on. The result is a series of reflections travelling back and forth along the transmission line each of which is successively smaller than the last. It is these reflections that cause ringing.
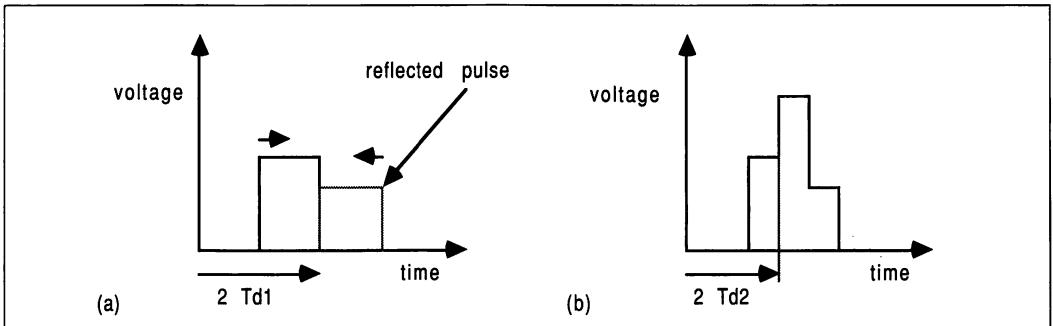
Figure 2.6 Simple reflections

Figure 2.6 shows a simplified picture of the effect of a reflection on the transmitted signal. Figure 2.6 (a) shows the waveform of the transmitted signal with the length of the transmission line at the critical length when the round trip delay (2Td1) is long enough to prevent the reflected waveform interfering with the transmitted waveform. In this case, ρ has a value of two thirds, the reflected pulse has a magnitude two thirds that of the transmitted pulse, shown in dotted lines, travelling in the opposite direction. Figure 2.6 (b) shows the effect of the reflection interfering with the transmitted pulse where the round trip delay (2Td2) in this case is sufficiently small. If the load has a reactive impedance, the resulting waveform will exhibit capacitive and inductive effects. If the load is inductive, it will initially behave as an open-circuit, finally behaving as a resistance. Alternatively, a capacitive load will initially behave as a short circuit, then finally acting as a resistance. These effects will result in the reflected waveforms having time constants.

Controlling transmission line effects

Ringing and undershoot are undesirable because they reduce the system noise margin. Some method of minimising undershoot is required. This is achieved by correct termination i.e. matching the impedance of

the transmitter and/or receiver to the characteristic impedance of the transmission line. A simple method of termination that requires no dc power is *series termination*.

A resistor is placed in series with a transputer LinkOut pin such that the combined impedance of the resistor and the output impedance of the link pin is equal to the characteristic impedance of the transmission line. The resulting transmission system is shown in figure 2.7.
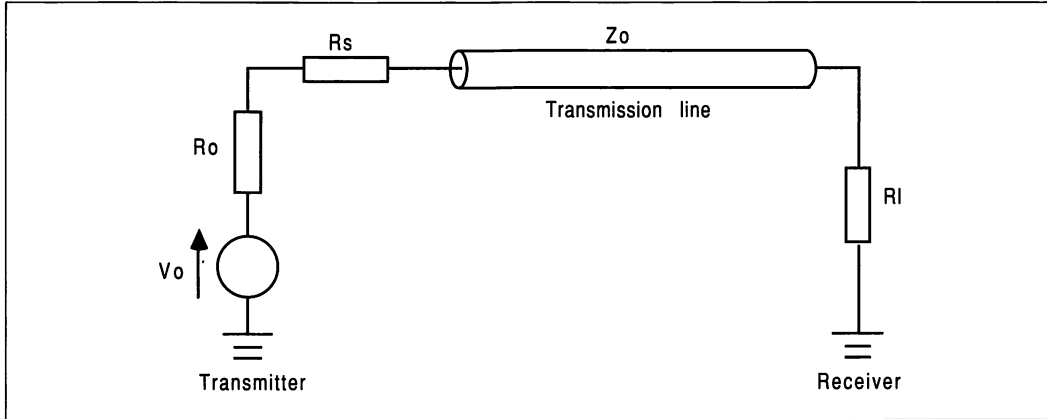
Figure 2.7 Series terminated transmission line

If $R_l > Z_o$, the reflection coefficient at the load is

$$\rho_l = \frac{R_l - Z_o}{R_l + Z_o} \quad and \quad 0 < \rho_l \leq 1$$

If $R_o + R_s = Z_o$, the reflection coefficient at the source is

$$\rho_s = \frac{(R_o + R_s) - Z_o}{(R_o + R_s) + Z_o}$$
$$= 0$$

This means that a transmitted signal will be reflected at the receiver, but the termination resistance will absorb the reflection, thus preventing any further reflections from reaching the load.

A single specified value of resistor will not be able to match the link output in all cases. The on-resistances of the P and N transistors of the link output are different and also vary between devices, with temperature and with supply voltage. Thus, a matching resistor may be specified to cope approximately with most variations.

Unless the transmission line is very well matched, the propagation delay down the line should not exceed 0.4 of the bit period at the operating link speed. Owing to the operation of the link output pad, a reflection arriving at the link output pin during a logic transition may cause a glitch on the local power supply of the link, possibly corrupting data.

The oscilloscope plot in figure 2.8 shows a data byte transmitted at 10Mbits/s over 24 metres of 100Ω characteristic impedance twisted pair cable with no termination resistor. The top trace shows the waveform at the LinkOut pin. The reflections can be seen to arrive back at the sending end at a time twice the propagation

delay later. The trace at the LinkOut pin is attenuated due to the effective potential divider caused by the resistances of the link output pad and the line. The dotted line shows the trace of the waveform, had there been no reflections. Thus, the reflections can be seen to be summated with the sending waveform, shown by the peaks on the data bits. The irregularity of the waveform is caused by the reactive load, discussed earlier in this note.

The bottom trace shows the received signal at the other end of the cable. Note the overshoot on the falling edges of the data bits caused by the signal being reflected a second time at the source.

The link interface inputs data by sampling each data bit 5 times, the correct value of the data being deduced as a result of these samples. Thus, excessive ringing may cause incorrect bit samples to be taken, corrupting data.
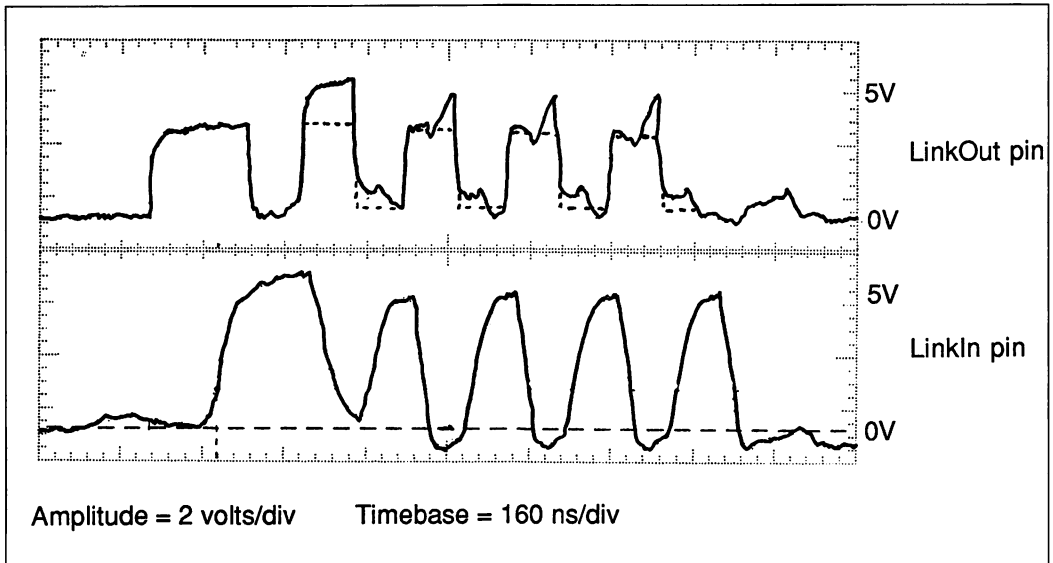
Figure 2.8 Reflections on a data packet

The plot in figure 2.9 shows the effect of inserting a resistance of 49Ω between LinkOut and the cable. The top trace shows that a reflection occurs at the receiver which travels back to the transmitter, in a similar manner to that shown in figure 2.6. However, in this case the termination resistor absorbs the energy of the reflection, eliminating a second reflection. The overshoot on the received signal, as shown in the bottom trace, is now eliminated. Since data will be switching between 1 and 0 regularly, there is a tradeoff between minimising overshoot and overdamping the signal. The value of the resistor required should be approximately 56Ω.

Series termination has advantages over other forms of termination (e.g. parallel termination). No power supply other than the logic supply is needed and the overall power requirement is low. Distributed loading along the line cannot be used, but since links are used point-to-point this is not a problem.

The link cables supplied with INMOS board products are made from twist 'n' flat cable. This is 28 awg twisted pair cable with 2-inch flat sections every 18 inches to provide easy connector termination. The nominal characteristic impedance of this cable is 105Ω. A 56Ω series termination resistor provides good matching between the transputer and the cable.

Amplitude = 2 volts/div   Timebase = 160 ns/div
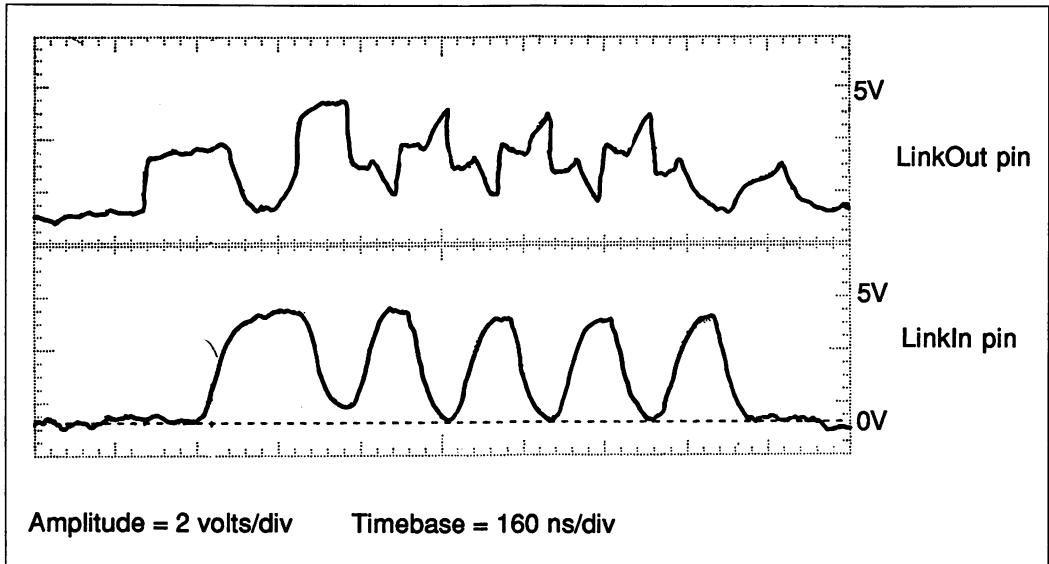
Figure 2.9 Data packet with a matched line

2.3.2   Noise and crosstalk

Noise or electromagnetic interference (EMI) can come from numerous sources including lightning, electrical machinery and electrostatic discharges, any of which can cause interference on a communications line. Link signals are TTL compatible and as such have a specified noise margin when directly driving a TTL input:

$$V_{OH}(MIN) - V_{IH}(MAX) \quad = \quad 2.4 - 2$$
$$= \quad 0.4V$$

$$V_{IL}(MIN) - V_{OL}(MAX) \quad = \quad 0.8 - 0.4$$
$$= \quad 0.4V$$

i.e. noise on the line must be limited to 0.4V in order to avoid the possibility of unwanted changes in logic level.

Crosstalk occurs when signal lines are run close together. The changing signal in one line is coupled into the other line, appearing as a noise voltage which is proportional to the rate of change of the current in the first line, for inductive coupling. Noise produced by capacitive coupling is proportional to the rate of change of voltage.

The protection of electronic circuitry from noise is a large subject [4], but some simple steps can lead to a reduction in noise pickup and crosstalk. Using twisted pair cable having a ground return twisted with each link signal line helps to reduce *differential mode* noise, i.e. noise which appears between the link signal and ground. Figure 2.10 shows the connections of an INMOS standard link cable. Note how each link signal line has its own ground. This also helps maintain a constant characteristic impedance along the cable.

Screened twisted pair increases the immunity from *common mode* noise, i.e. noise coupled equally into both wires in a pair. Crosstalk can appear as common mode noise, depending upon the construction of the
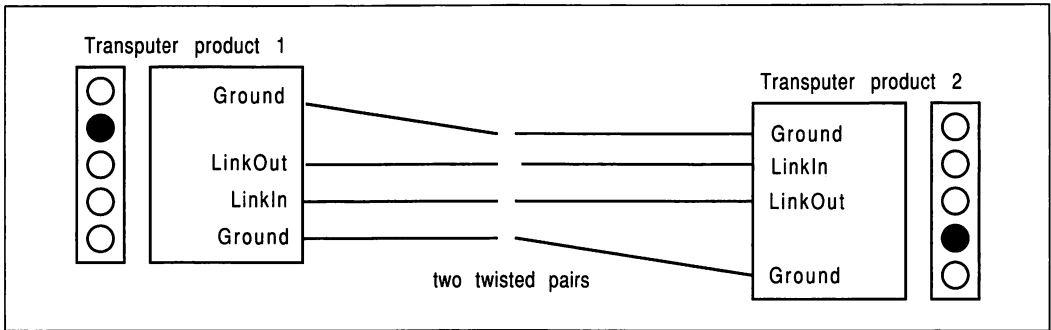
Figure 2.10 INMOS standard link cable

cable, and can be reduced by screening individual pairs. Figure 2.11 shows a test set up to record crosstalk between link signal lines. A process running on transputer T1 continuously sends the byte AA hex, i.e. bytes containing alternate '1's and '0's. Transputer T2 sends Acknowledge packets.
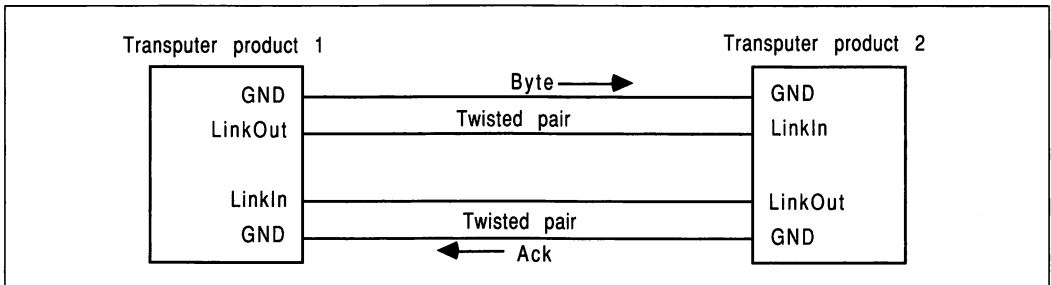
Figure 2.11 Crosstalk test

Figure 2.12 shows a plot of the crosstalk induced from the byte on T1 LinkOut0 onto T1 LinkIn0 when 10 metres of unscreened twist 'n' flat is used. The peaks at the extreme edges of the plot are the acknowledge start bits. These peaks have been clipped by the oscilloscope in order to show the crosstalk on a reasonable scale. The crosstalk is caused by the rapid edges of the data packet bits in the other signal wire. It can be seen that the data packets are transmitted between acknowledges on separate lines.

The measurements are taken when transmitting at 20Mbits/s with no series termination.

Figure 2.13 shows a similar plot using 18 metres of twisted pair where each pair is individually screened. Again, the two large, clipped peaks at either side of the plot are the acknowledge start bits with the data packet crosstalk being shown between the acknowledges. The dotted line on the inset trace shows the (exaggerated) waveform of the data packets on the other signal wire in order to show the correlation between the edges of the data packet and the crosstalk being coupled onto the other signal line.

The crosstalk is reduced from 1.77V to 760mV peak to peak, a reduction of more than 7dB due to the screening. A similar performance can be expected for external noise rejection. Since noise pickup increases with the length of line it is recommended that long links implemented with twisted pairs are screened. An overall screen is adequate, but individually screened pairs will improve the rejection of crosstalk. Screens should be connected to the frame ground at both ends of the cable, due to the high frequency components in the edges of the data.

Amplitude = 500 mvolts/div   Timebase = 200 ns/div
Peak - to - peak = 1.77 volts
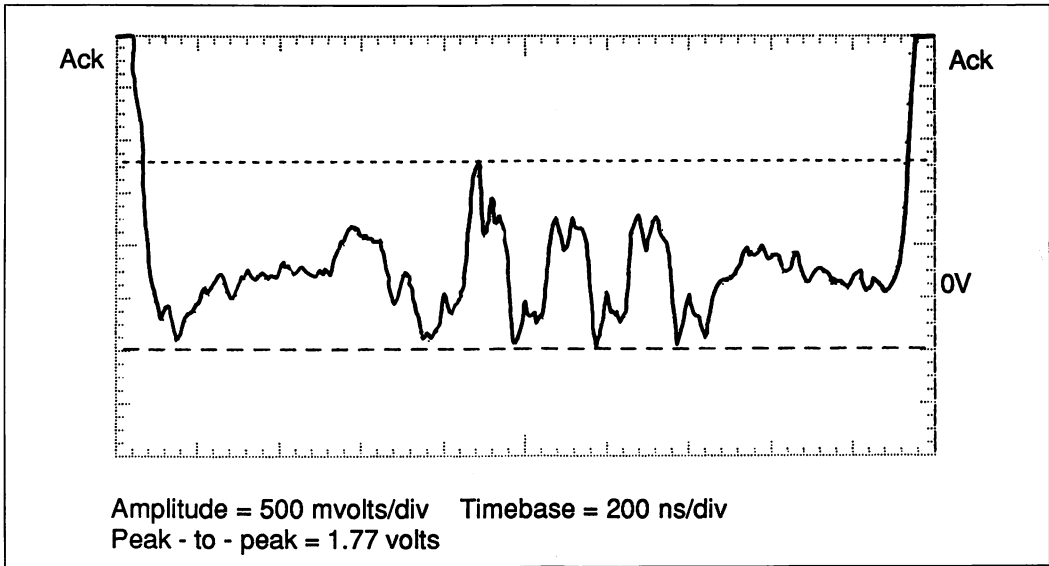
Figure 2.12 Crosstalk on a 10m twisted pair

Amplitude = 200 mvolts/div   Timebase = 150 ns/div
Peak - to - peak = 760 mvolts
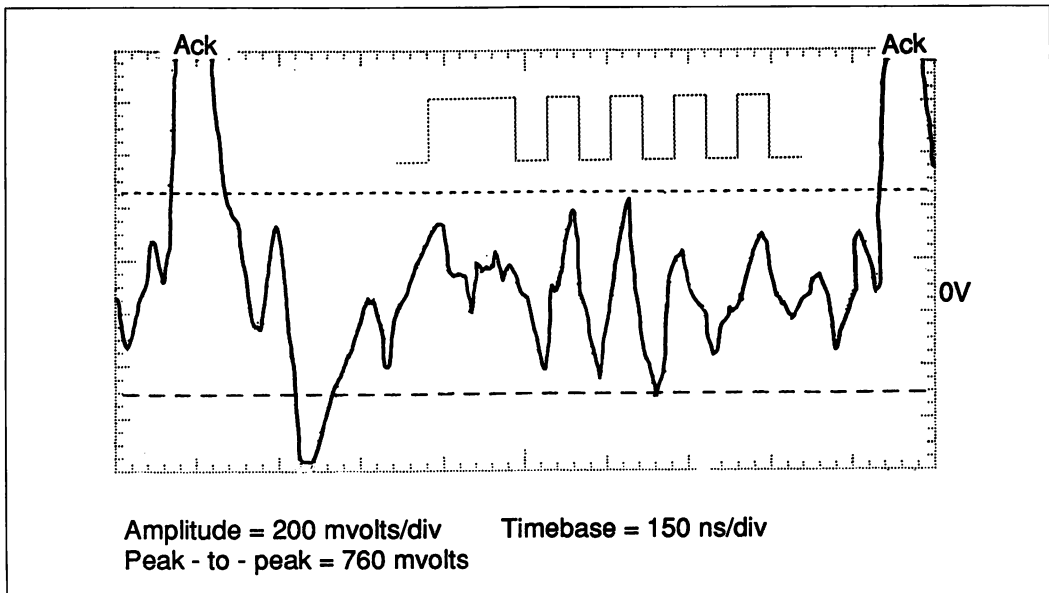
Figure 2.13 Crosstalk on a 18m screened twisted pair

2.3.3   Differential line drivers/receivers

Differential line drivers/receivers such as EIA Standard RS 422 [5], when used with twisted pair, provide maximum noise immunity. Because the signal is sent differentially common mode noise is rejected by the receiver up to its common mode rejection limit. Figure 2.14 shows an implementation of an RS 422 system suitable for use with INMOS links. This system has been used by INMOS for reliable link transmission over

160 metres of twisted pair at 5Mbits/s. It should be noted that the RS 422 specification limits the maximum bit rate to 10Mbits/s at a maximum distance of about 15 metres using 24 awg twisted pair. At 5Mbits/s, the maximum length is about 25 metres. The RS 422 specification is, however, deliberately conservative.
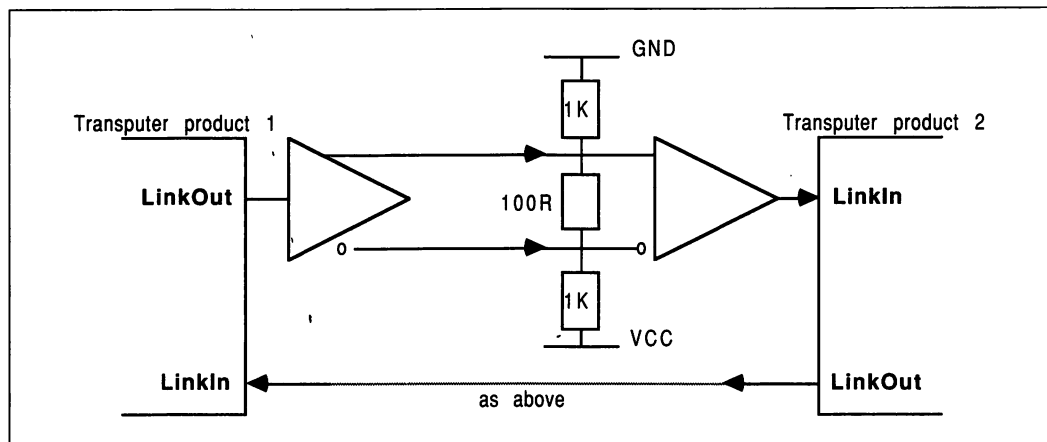
Figure 2.14 RS 422 link

2.3.4   Attenuation

Assuming a noise free environment, the maximum length of line over which a link signal may be transmitted without buffering is determined by the attenuation of the line. Attenuation of twisted pair increases with the frequency of the signal transmitted along it. The bandwidth required for transmission of the significant frequency components of the link signal line spectrum can be expressed by

$$f_{3dB} \quad = \quad \frac{350}{6(ns)}$$
$$= \quad 58\text{MHz}$$

Where f_{3db}= the frequency at which the line spectrum components are decreased by 3dB.i.e. 50% of the initial magnitude, assuming a minimum fall time of 6ns.

This arises from the fact that high frequency components are attenuated more than low frequency ones, resulting in slower edges and the height of the corners of a signal being reduced.

For a maximum signal reduction of 0.4V from the logic 1 level, the permissible attenuation is

$$
\begin{aligned}
dB &= 20\log\left(\frac{V_1}{V_2}\right) \\
&= 20\log\left(\frac{2.4}{2}\right) \\
&\simeq 1.6dB
\end{aligned}
$$

The maximum line length is then

$$
l_{max} = 100m\left(\frac{1.6}{\text{Atten}}\right)
$$

where Atten is the cable attenuation in dB/100m at the operating frequency. For example, using twisted pair with an attenuation of 30 dB/100m at 58MHz

$$
\begin{aligned}
l_{max} &= \frac{100m\text{x}1.6}{30} \\
&= 5.3m
\end{aligned}
$$

This value is of course the maximum length of cable which will allow all frequency components up to 50MHz. The received signal will still be adequate, regardless of the rounding effects of the low pass filtering action of the cable. From figure 2.5, a link with a capacitive load of 80pF will have a fall time of 10ns. This corresponds to a maximum frequency component of 35MHz. For a cable with an attenuation of 18dB/100m at 35Mhz, the maximum length of cable is 8.9m.

2.3.5   Buffering

If longer links are required buffer/line drivers may be used (figure 2.15). Because of the asynchronous operation of links, the round trip propagation delay is unimportant as far as reliability is concerned.It is, however, important that the skew introduced by the buffers is less than the maximum skew quoted in the Transputer Reference Manual [1]. (Skew is discussed further in the next section.) To minimise skew and to maximise noise margin at all link speeds it is recommended that FACT buffers are used [6], e.g. the 74AC244 octal buffer/line driver.
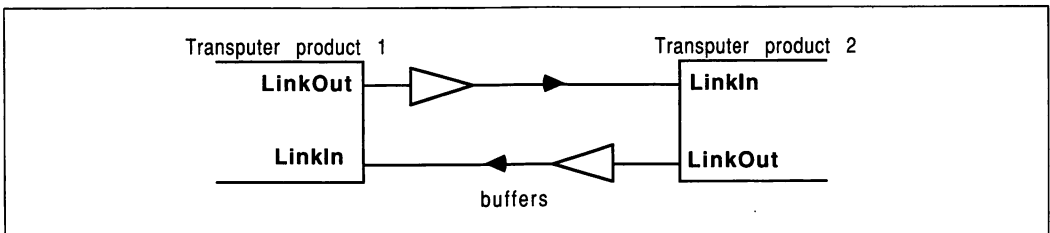
Figure 2.15 Buffered links

At Vcc = 4.5V

Voh = 4.4V

Vih = 3.15V

Therefore

$$\text{Attenuation} \quad = \quad 20\log\left(\frac{V_{OH}}{V_{IH}}\right)$$
$$= \quad 20\log\left(\frac{4.4}{3.15}\right)$$
$$= \quad 2.9\text{dB}$$

Hence

$$l_{max} \quad = \quad \frac{100m\text{x}2.9}{18}$$
$$= \quad 16m$$

assuming the same cable as previous examples.

While the FACT data book states that the input and output diode clamps on a FACT device will match most transmission line impedances, it is recommended that a series matching resistor is used at the buffer output. The series resistor should be equal to the characteristic impedance of the transmission line. Figure 2.16 shows a plot of a bit taken at LinkIn in figure 2.15, operating at 10Mbits/s along a 50cm INMOS link cable. No termination resistor is used and ringing results. Figure 2.17 shows a similar plot taken after the insertion of a termination resistance of 91Ω which damps the ringing.

Amplitude = 2 volts/div   Timebase = 20 ns/div
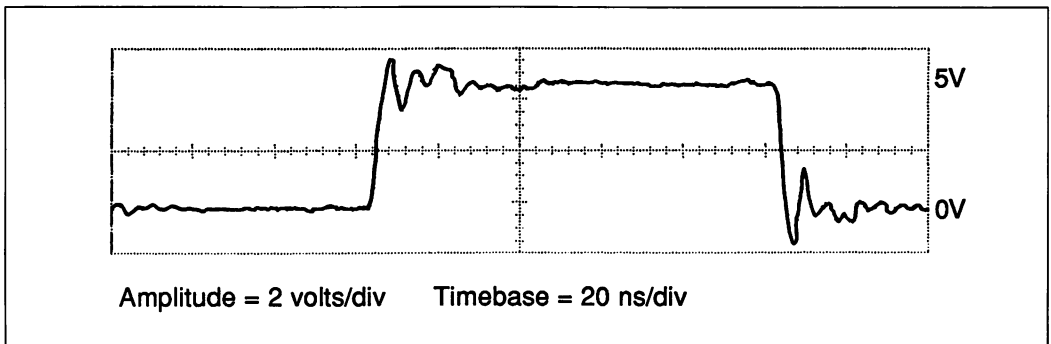
Figure 2.16 Ringing at FACT buffer output

Amplitude = 2 volts/div   Timebase = 20 ns/div
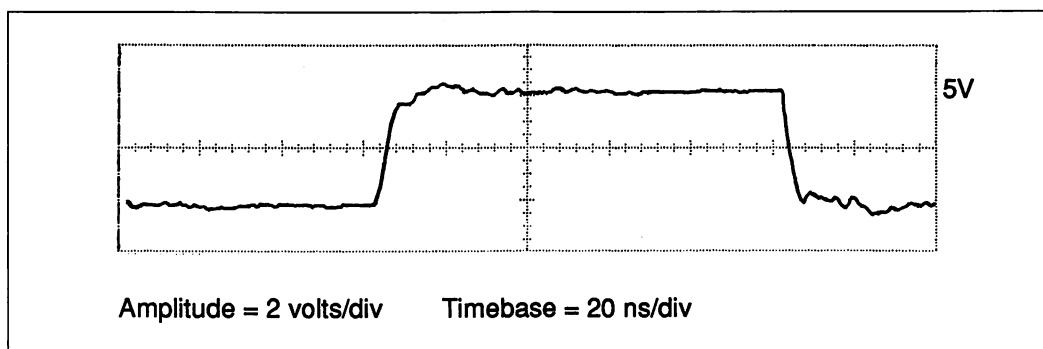
Figure 2.17 Series damped FACT buffer output

2.3.6   Skew

The *skew* of a system is defined as

$$\text{skew} = \max \left\{ |t_{PLH} - t_{PHL}| , |t_{PLHi} - t_{PHLj}| \right\}$$

where t_{PLH} is the system propagation delay for low to high signals, and t_{PHL} is the propagation delay for high to low signals . The rising edge of a start bit is denoted by t_{PLHi} and t_{PLHj} relates to successive rising edges. The effect of skew is to broaden or narrow digital signals in the system. This changes the times at which data bits and the stop bit (and the next start bit) are seen at the receiving end relative to the leading edge of the start bit, shown in figure 2.18.

Skew varies instantaneously with power supply variations.
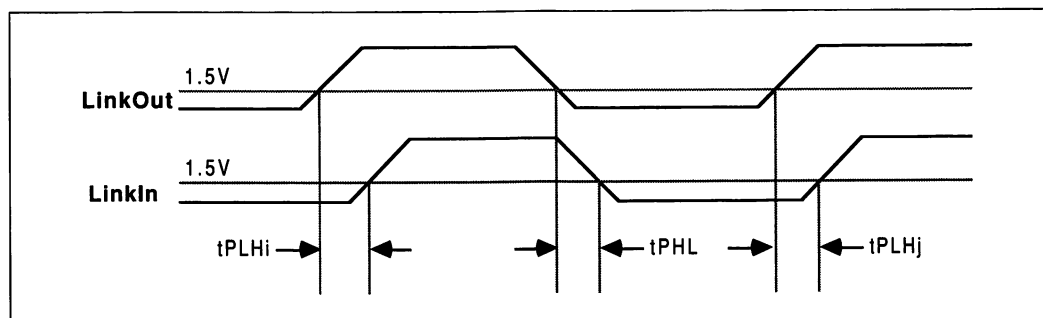
Figure 2.18 Skew

Figures 2.19 and 2.20 show some of the causes of skew. Figure 2.19 demonstrates how skew is introduced by buffering link signals. The skew arises as a result of the buffer exhibiting differing propagation delays for rising (t PLH) and falling (t PHL) edges, thus distorting the pulse width and reducing the sampling window. Skew of this nature can be largely eliminated by using FACT buffers which exhibit relatively little skew.

Figure 2.20 shows the effect of having independent grounds for each link interface. Small changes in the voltage between the separate grounds can cause ambiguous data samples. This diagram also shows the effect of a voltage caused by noise on the link data. Instantaneous voltages of this nature may also result in incorrect samples, hence the need for adequate noise control.

While the overall propagation delay of a line has no effect on the reliability of a link, there is a maximum amount of skew that the link interfaces can withstand before they fail. Table 2.1 shows the absolute maximum value of skew obtained experimentally that links can withstand at the three link speeds. These figures were obtained in an environment designed to be harsh by omitting a ground plane and decoupling capacitors.
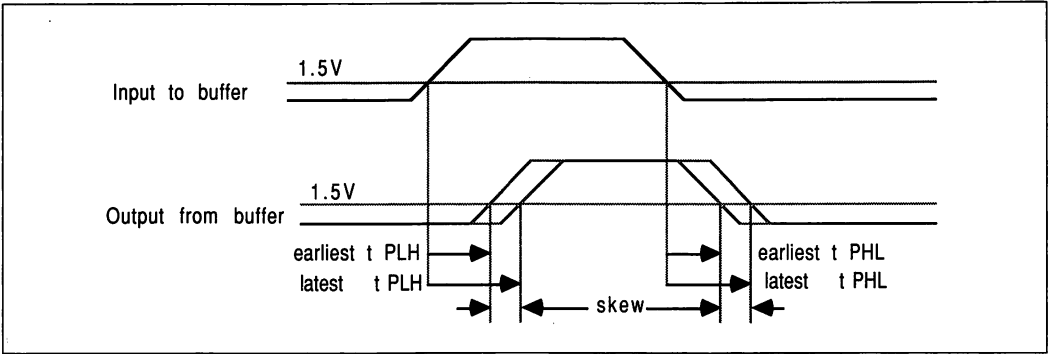
Figure 2.19 Skew caused by buffering
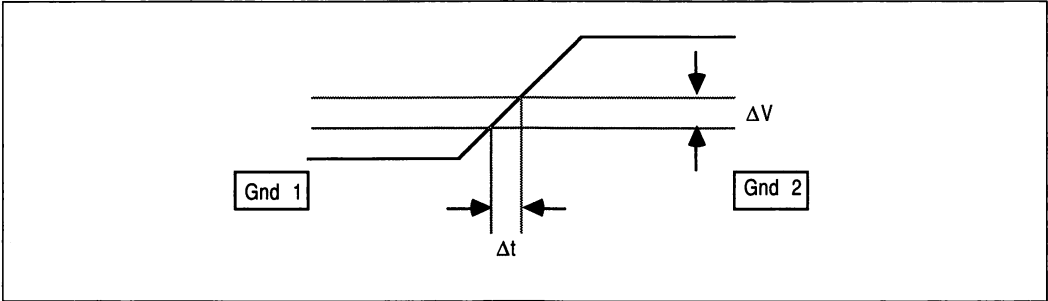
Figure 2.20 Other causes of skew

Link speed (Mbits/s)	Bit period (ns)	Max skew (ns)
5	200	40
10	100	15
20	50	8
Vcc = 5V, skew measured at 1.5V		

Table 2.1

This does not imply that the maximum skew figures quoted in the transputer reference manual [1] should be exceeded.

2.3.7   Protection of links

In order to protect links from electrostatic discharge (ESD) the circuit shown in figure 2.21 is used. The circuit is required for each LinkIn pin. The Schottky diode protects the link from ESD up to 2kV, while the resistor prevents the link input from floating high when not in use. The diode also helps to eliminate overshoot on received link signals by turning on when LinkIn rises more than about 0.4V above Vcc.
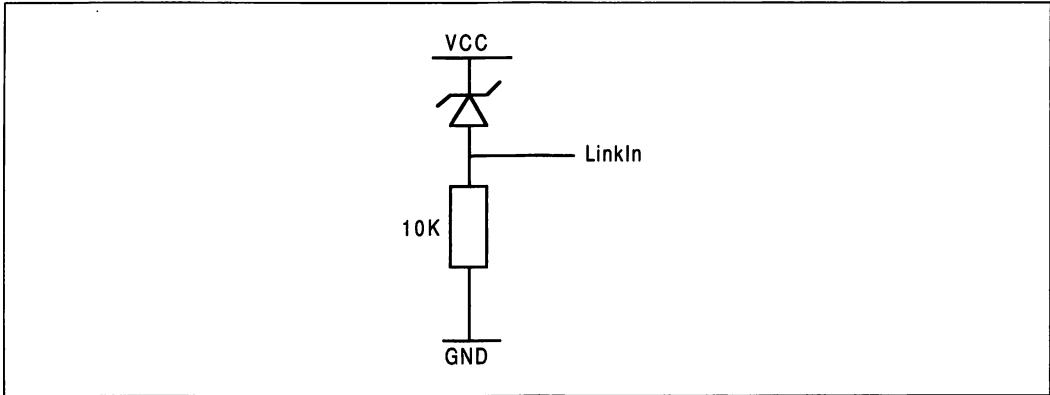
Figure 2.21 Link protection

Figure 2.22 shows a plot of a bit received at a LinkIn. Note how the clamping effect of the diode eliminates any overshoot on the leading edge of the pulse. With the addition of another diode (figure 2.23) the circuit can be used to terminate a transmission line. The diodes clamp signal overshoot in both directions.

Amplitude = 1 volt/div   Timebase = 50 ns/div
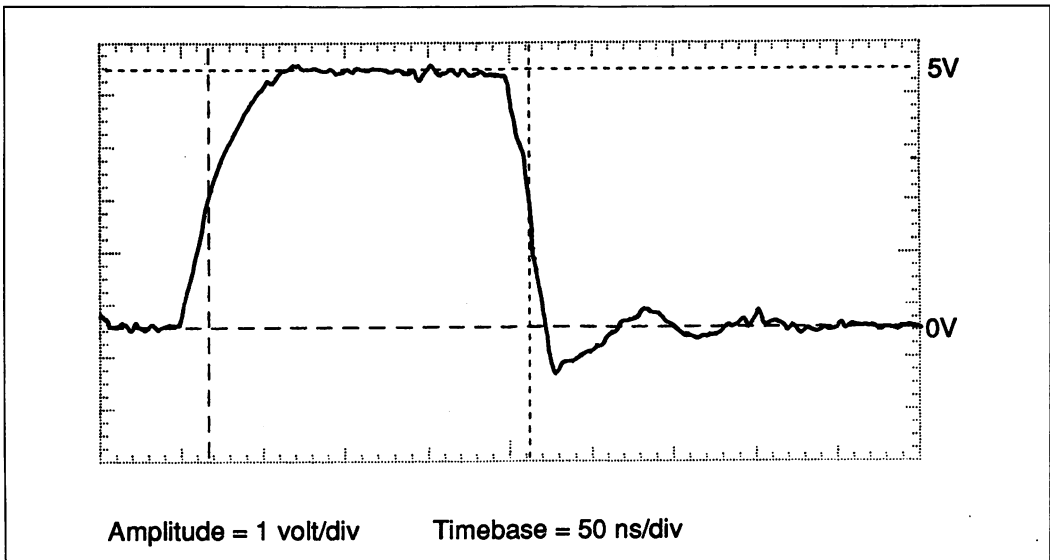
Figure 2.22 Clamping effect of a diode
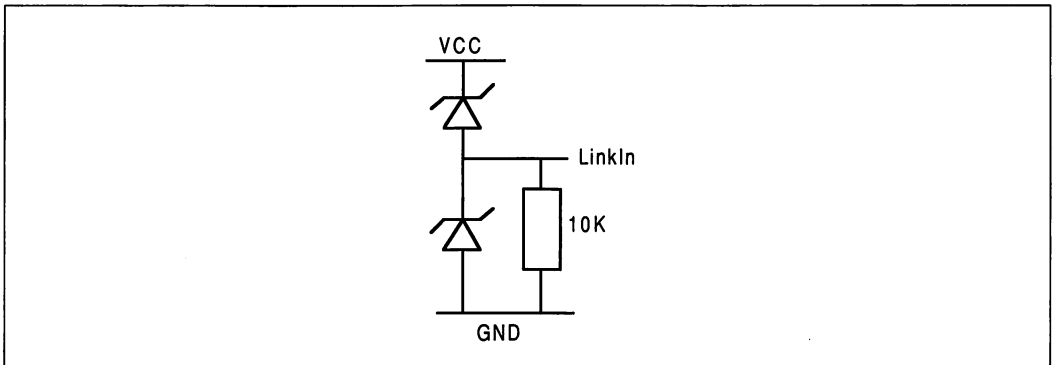
Figure 2.23 Schottky diode termination

2.4   Implementing an INMOS link using optical fibres

When operating over short distances, e.g. within an ITEM, standard twisted-pair link cables provide a reliable link medium at all link speeds (5 , 10 and 20 Mbits/s). Over longer distances, however, reliable transmission is affected by the characteristics of the line .i.e. Attenuation, pulse distortion and noise susceptibilty.

One method of overcoming these disadvantages is to use an optical fibre. It is not the intention of this application note to educate the reader in all aspects of optical fibres. The purpose of the note is a simple discussion of the issues that arise when engineering an INMOS link, using optical fibres.

2.4.1   Advantages of optical fibres

Optical fibres have a very high bandwidth, greatly reduced attenuation and are physically very light compared with more conventional media e.g. coaxial cable.

Optical fibres exhibit no susceptibility to crosstalk or external noise.

Owing to the total electrical isolation offered by optical fibres there is no danger of ground current loops and ground noise being coupled between individual systems.

An optical fibre system is inherently difficult to tap onto. This makes it almost impossible for a third party to monitor information being transmitted on an optical fibre without being detected.

2.4.2   An implementation of a 5 Mbits/s INMOS link using optical fibres

The INMOS link is an asynchronous means of sending data between transputer family devices.

Although the link was originally designed for local communication, communication over longer (> 100m) distances is best achieved by using optical fibres.

Because of the asynchronous nature of the INMOS link protocol, the propagation delay of the link does not affect *reliability*, it may, however, affect *performance*. The time delay between a transputer device sending a data packet and receiving an acknowledgement increases with the length of the link, thus decreasing effective data throughput.

The time taken to send a data packet and receive the corresponding acknowledge can be expressed by:

$$T_{tot} = T_{dp} + T_{ap} + 2lT_{mpd}$$

where

l is length of the link
T_{dp} is the time taken to output a data packet
T_{ap} is the time taken to output an acknowledge
T_{mpd} is the propagation delay of the transmission medium per unit length

The following graph (figure 2.24) shows plots of maximum data throughput at the various link speeds versus length of optical fibre using the link implementation provided by a transputer family device such as the T414 or T212.

It can be seen that the difference in data throughput at different link speeds decreases with increasing length of optical fibre, the main contributing factor to the delay being the propagation delay of the medium, the constant hardware overheads becoming negligible.

It can be seen that for a medium of length approximately 500m, the effective difference in performance for the various link speeds is very much decreased. Therefore, at longer fibre lengths, there is very little advantage to be gained by operating the links at 20 MBits/s rather than, say, 5 MBits/s. This fact allows the designer to relax the constraints (especially skew) of the system.

However, the graph does not strictly apply to the link implementation provided by the T800 or T222. Increased performance is provided by such links over longer distances due to the overlapped acknowledges.
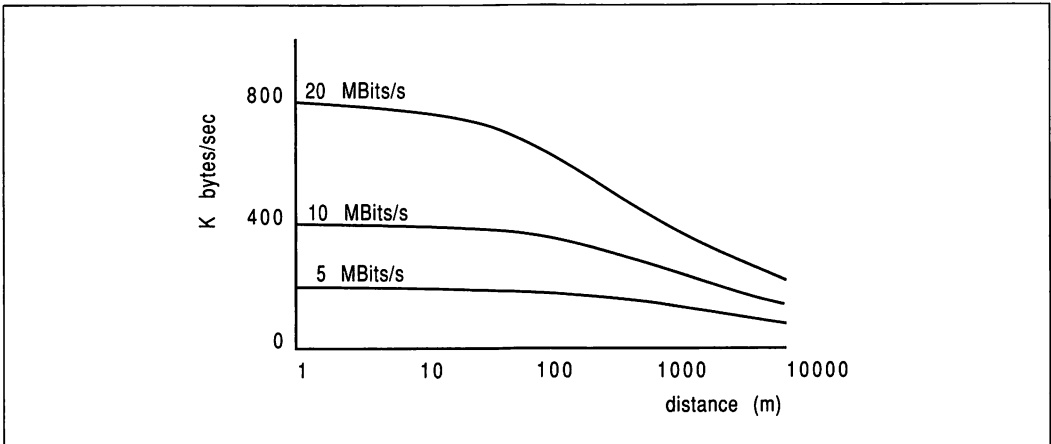
Figure 2.24 Effect of link length on data throughput

Fibre bandwidth considerations

However, even with optical fibres, there is a limit placed on the maximum length of fibre owing to the skew restrictions of the INMOS link inputs [1]. This skew is caused, in the case of optical fibres, by the phenomenon known as dispersion.

There are two basic causes of dispersion, chromatic and modal. Chromatic dispersion arises from light of different wavelengths propagating at varying velocities through the fibre. Modal dispersion is caused by reflections at the interface between the core and the cladding of the fibre. This results in the reflected wave having a longer effective path length than a wave propagating directly through the fibre with no reflections.

Owing to the difference in path lengths, the optical signals will not arrive at the receiving end of the fibre at the correct moments in time, resulting in dispersion.

Different types of fibre exhibit varying dispersion characteristics, offering the optical system designer a range of price/performance tradeoffs. However, problems with dispersion will tend only to occur at much longer distances. (> 1km)

The recommended maximum skew across the system is 30ns for the 5 Mbits/s linkspeed compared to the 20 Mbits/s tolerance of 3ns. A fibre used at low data rates can have a higher dispersion without affecting link reliability, owing to the increased skew tolerance.

Choosing a fibre

When constructing a system, parameters such as attenuation, dispersion (modal and chromatic) and bandwidth must be considered when choosing an optical fibre. Speed of data transmission, skew tolerances and maximum length of fibre are determined by the characteristics of specific fibres and transmitter/receiver components.

For example, laser devices will cause less dispersion than light-emitting-diode type devices.

Graded index fibres will decrease modal dispersion.

Monomode fibres will largely eliminate modal dispersion.

For further information consult reference [7].

Flux budgeting

An optical fibre system consists of a transmitter, fibre and receiver. The technique of ensuring sufficient optical power is transferred through the system to drive the receiver correctly is known as *flux budgeting*. Each component in the system will have an associated power loss. The maximum length of any optical fibre system can be calculated using the following equation:

$$P_t - al >= P_r + M_p$$

where:

       P_t = transmitter power(dBm) measured at the end of 1m of fibre
       a = fibre attenuation per length(dB/km)
       l = cable length(km)
       P_r = minimum optical power required by the receiver
       M_p = optical power margin set by user (>1 dB)

Recommended components

This note is intended to give the reader some idea of the method of implementing an optical fibre link.

For evaluation purposes a simple circuit was constructed. The devices to be used were required to be relatively inexpensive, simple to use and to comply with the constraints of the INMOS link engines. Of the devices considered, those manufactured by Hewlett-Packard were found to be suitable. Those used were:

Transmitter : HFBR 1402

Receiver : HFBR 2402

The transmitter is an 820nm Gallium Arsenide light-emitting-diode and the receiver is a PIN [7] photodiode with a TTL compatible output.

These devices simply plug directly into a printed circuit board, require a minimum of support circuitry and are

fully TTL compatible.

The devices are fitted with the emerging industrial standard for optical fibre connectors, the SMA connector. This enables a fibre previously fitted with SMA connectors to be screwed directly onto the device, allowing simple interchanging of fibres. At present, these devices will only operate reliably at speeds up to 5 Mbits/s. It is expected that equally suitable components enabling higher data rates will be available in the not too distant future.

The major advantage of these devices is the fact that they are DC coupled i.e. there is no requirement for a steady stream of data passing between transmitter and receiver as is found in the more common AC coupled devices. AC coupled devices tend to require minimum data rates and impose restrictions on the duty cycle of data being transmitted. Devices of this nature are obviously of no use for link communication unless some method of encoding and perhaps having to send dummy packets is incorporated into the circuit, thus increasing circuit complexity. Such methods tend to move away from the idea of simple communication, provided by the link itself. For more information consult reference [8].

For our evaluation purposes the fibre used was 200 PCS (Plastic Clad Silica), a step index fibre [7]. This method of construction exhibits greater attenuation and dispersion than graded index fibres. However, this problem is offset by the ability of PCS to couple more optical power between transmitter and receiver.

Transmitter circuit

Figure 2.25 shows the circuit required to operate the transmitter. As can be seen, the transputer family device link output is simply directly connected to the input of the circuit. No driver circuitry is required in this case as the link output provides sufficient current to drive the optical transmitter. However for more extreme cases (e.g. longer distances or higher attenuation fibre) the LED may require more drive current in order to provide the receiver with sufficient optical power.
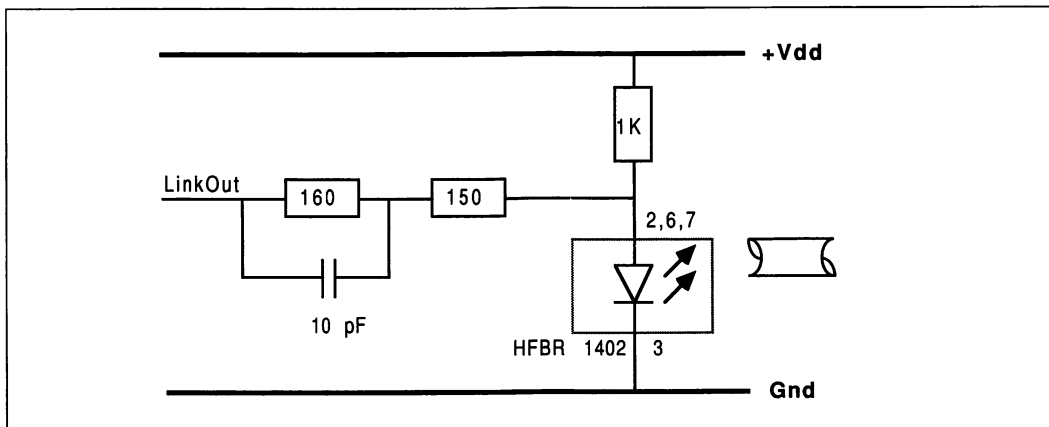
Figure 2.25 Transmitter circuit

The component values shown are calculated from the equations given in [8] for a drive current of 25mA. The 10pF capacitor is a 'speed-up' capacitor, intended to square the edges of the input signal.

Receiver circuitry

The receiver is an open-collector device, requiring a pull-up resistor. Owing to the nature of the operation of a photodiode, the incoming logic value is inverted at the output. It must be stressed that, in order to invert the receiver output signal, a FACT inverter should be used. The FACT technology provides very fast edges, with negligible skew, making FACT an ideal logic family for interfacing with INMOS links. A suitable device is the AC04 hex inverter IC. The output of the FACT buffer is connected, by methods described earlier in this note, to the input of a link.
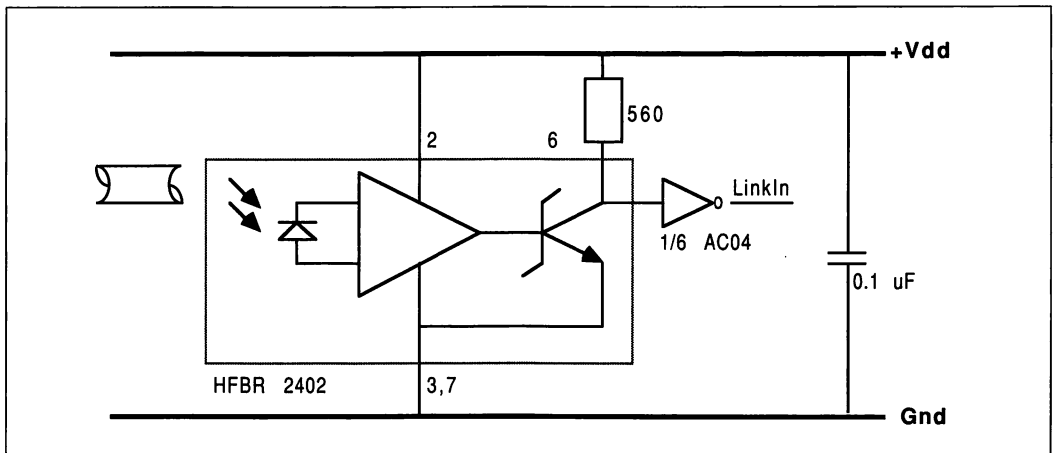
Figure 2.26 Receiver circuit

Physical considerations

It must be stressed that, although optical fibres offer many advantages over conventional wire, they cannot be treated as such. Multiple fibres may be contained within a single sleeve, allowing easy installation of numerous links. In applications requiring multiple connections (e.g. an ITEM module) allowance must be made for the extra space required for the fibre bending. A typical fibre has a minimum bend radius of 2.5 cm. In having multiple link connections using such devices as those supplied by Hewlett-Packard there is a problem concerning board area, as two devices are required for each link. This allows a small maximum number (approximately 3–6) of links to be realised on the edge of a double Eurocard. The devices can be placed away from a card edge. However, this increases the difficulty of repeated connection in multiple card systems.

One way of circumventing the problem of a fixed amount of optical fibre links is to use the IMS C004 [9]. This effectively allows dynamic reconfiguration of up to 32 INMOS link inputs to be connected to up to 32 outputs. This device allows the network of transputers to be reconfigured, thus allowing the optical links available to be shared between different devices on a board or in a system, giving a more efficient use of board area at slightly increased circuit complexity .

Conclusion

The INMOS link, by the very nature of its operation, demands a minimal delay between sender and receiver via a noise free medium. Optical fibres are able to provide such a medium and over longer distances than conventional methods.

The optical fibre electrically isolates separated sytems.

The simplicity and low cost of implementing a 5 Mbits/s INMOS link with optical fibres has been demonstrated in this note, using the devices produced by Hewlett-Packard.

It is beyond the scope of this note to discuss all aspects of optical fibre system design. For high performance systems the reader should consult reference [7].

2.5   Summary

Although links were originally designed for local communications between devices on a pcb or across a backplane, it is possible to use them over longer distances. However, some precautions must be taken to ensure reliability and integrity of data, as summarised below.

Distance	Method of connection	Comments
Up to 30cm	Direct connection	Suitable for pcbs,backplanes
Up to 10m	Series termination	56Ω to match 100Ω transmission line
Up to 20m	FACT buffers	Minimal skew
Up to 30m	RS 422	Suitable only for 5 or 10 Mbits/s. Good noise immunity
Over 30m	Optical fibre	Noise free, low attenuation. 5Mbits/s system demonstrated Simple engineering over long distances

2.6   References

1 *Transputer Reference Manual.* INMOS Limited. 1986

2 *occam reference manual.* INMOS Limited. 1986

3 *IMS T414 Data Sheet.* INMOS Limited. 1986

4 *How To Control Electrical Noise.* Mardiguian, M. Don White Consultants, Inc. 1983

5 *EIA Standard RS 422A.* Electronic Industries Association. 1978

6 *FACT Data Book.* Fairchild Camera and Instrument Corporation. 1985

7 *Fiber Optics Handbook.* Hewlett-Packard. 1983

8 *Optoelectronics Designer's Catalog.* Hewlett-Packard. 1986

9 *IMS C004 Data Sheet.* INMOS Limited. 1987

3 IMS B003 design of a multi-transputer board

3.1 Introduction

The B003 evaluation board is a double extended Eurocard containing four T414 transputers, each with 256 Kbytes of dynamic RAM. The four transputers are configured in a square, and two links from each transputer are brought to the edge conector.

The interface from the B003 is via a 96 way DIN 41612 edge connector. Links 0 and 1 from each transputer are brought out via the edge connector together with the system services signals. The connector is a simple superset of the 64 way connector used by B001, B002 and other INMOS evaluation boards.

The board uses a minimum of glue logic. The system services shared by all the transputers consist of a single 5 MHz clock and three packs of TTL. Each transputer uses a further three packs of TTL to interface to its eight RAM chips. The minimal glue logic introduces minimal access time overhead for the RAM, and the T414-15 completes a memory access in four processor cycles.

The square connection of transputers makes it possible to test the board down a single link, minimizes edge connector pin count, and makes it possible to build a wide variety of networks. The application note bound with this note gives programmed examples of the B003 in a ring, a rectangular array, a 'butterfly' network (folded binary structure) and a hypercube.

3.1.1 Logic for each transputer

The logic for each transputer with its 256 Kbytes is shown in figure 3.1. The RAM is provided by eight 64K*4 dynamic RAMs, with just three TTL packs between the transputer and RAM. Apart from the RAM and TTL, there are a few discrete components for the links, for error, and for decoupling.
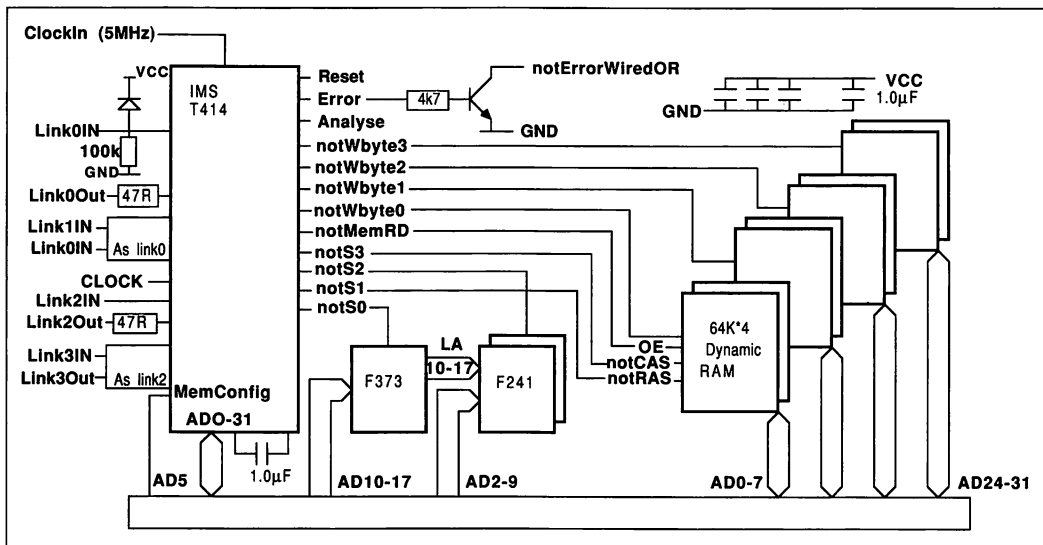
Figure 3.1 Logic for each transputer on IMS B003

Memory interface

The logic is used to latch the column address and to multiplex between the row address and column address.

The load on the F241 multiplexers is sufficiently small (50 pF), and the RAMs are sufficiently close to the F241 outputs that series matching resistors are not needed.

The control signals **notRAS**, **notCAS**, **notOE** and **notWEn** are taken directly from the transputer signals

notMemS1, **notMemS3**, **notMemRd** and **notMemWByten**. No buffering is needed because the transputers can easily drive the 50 pF load, and again no series matching resistors are required because the transputer is so close to the RAMs.

Using such a small amount of logic between the transputer and the RAM not only minimises cost, but also minimises delay. The RAM can therefore be used with minimal overheads on its access and cycle times. The timing diagram for the interface is shown in figure 14.2.6.

Starting **notRAS** at the earliest opportunities and latching the read data at the latest opportunity gives ample margin on access time from both **notRAS** and **notCAS**. Terminating **notRAS** early gives an adequate **notRAS** pulse width, and at the same time ensures sufficient precharge time.

Four processor cycles are used with the T414-15 and the 41464-12 RAMs because the cycle time of the RAM, at 220 ns, is more than would be provided by a memory interface cycle of three processor cycles.

Links
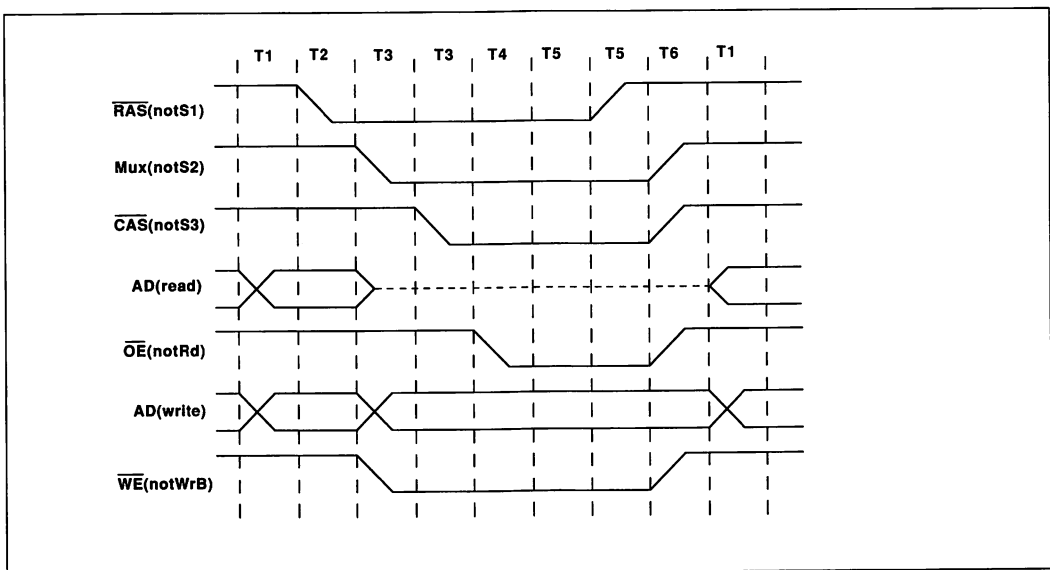
Figure 3.2 Timing diagram for memory interface

The links of the transputer used on the B003 are capable of running at 20 Mbits/s, at which speed they will not tolerate skew introduced by buffering.

Links 2 and 3 of each transputer which are connected within the B003 have a simple series termination on the **LinkOut signal**. The termination resistor of 47 ohms, combined with the output impedance of the **LinkOut** circuit, gives a termination impedance marginally below 100 ohms.

Links 0 and **1**, which are brought to the edge connector, also have 47 ohm resistors on the link outputs. The link inputs also need pull down resistors in case a link is not connected. On the transputers used on the B003, the link inputs are more sensitive to electrostatic discharge (ESD) than the link outputs, and so the link inputs which connect to the edge connector are protected by schottky diodes; with the diodes the transputer can withstand 'zap' tests of up to 2 kV without damage.

Error

The error output produced by the transputer is active high, which is suitable when there is one transputer on a board but causes extra wiring and logic if there are many transputers on a board. To simplify the wiring, a **notErrorWiredOR** signal is generated by a resistor and transistor.

Decoupling

The power supply decoupling for the RAM and for the TTL is so close to the transputer that it provides excellent decoupling for the transputer. In addition to the power supply decoupling a further capacitor is needed between CapPlus and CapMinus to decouple an internal power supply used by the phase locked loop/clock multiplier. This capacitor was originally a 10 μF tantalum capacitor, but has been changed to 1 μF ceramic for future production.

Printed circuit layout

The printed circuit is a straightforward 4 layer board with power and ground planes for the inner layers, and all signal traces on the outer layers. The design rules are an easily manufacturable 0.010" trace, with 0.008" between traces. Component pads are 0.070", with 1 mm holes; vias are 0.050" pads with 0.6 mm holes. Only one trace is allowed between pads.

The two outer layers are shown in figure 3.3.

(a) Component side   **(b) Solder side**
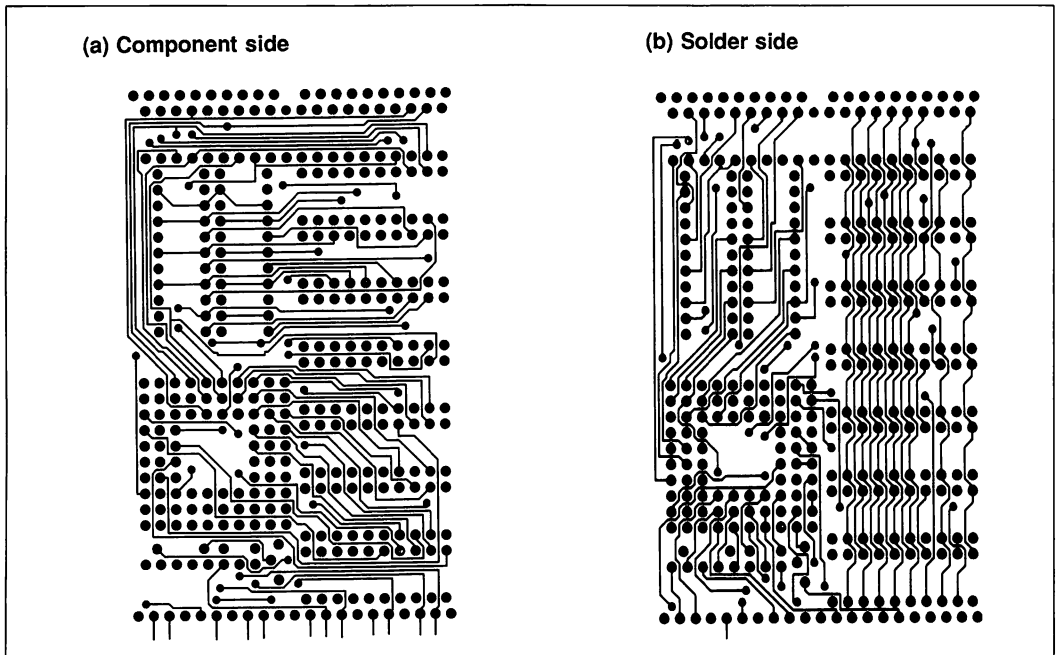
Figure 3.3 PCB layout: (a) Component side (b) Solder side

PGAs have been somewhat notorious for the difficulty they present to PCB layout. At first sight this layout appeared difficult, but careful component placement and orientation resulted in surprisingly simple layout, and there is still transparency for a number of additional connections.

Aspects of the placement which helped were:

- moving the link and control connections so they do not interfere with the memory connections;

- placing ICs lengthwise to the transputer. This allows maximum transparency, without pads getting in the way;

- moving the 373 to beyond the address multiplexors, which also had the effect of putting Byte 1 of the RAMs beyond Byte 0.

Overall signal flow is shown in figure 3.4:
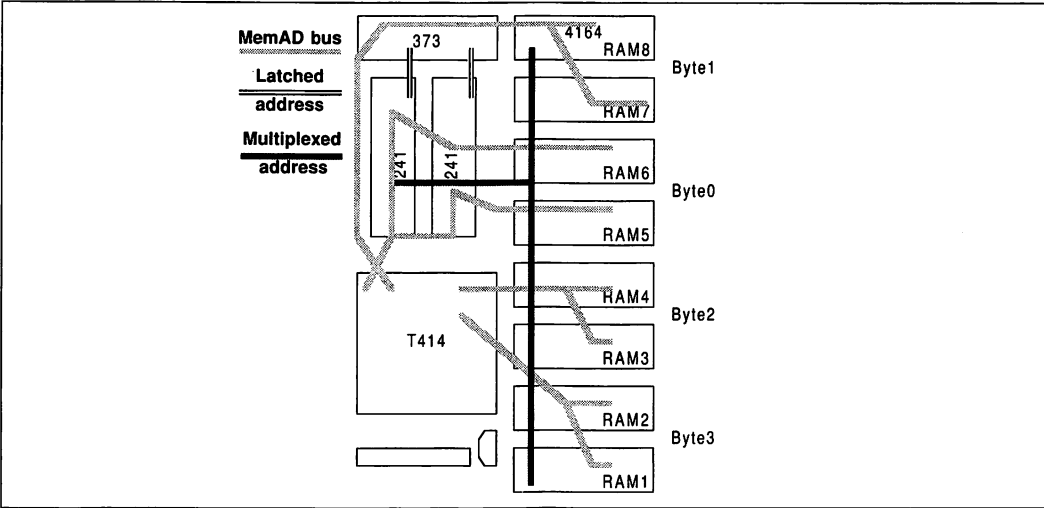
Figure 3.4 Signal flow on IMS B003 PCB
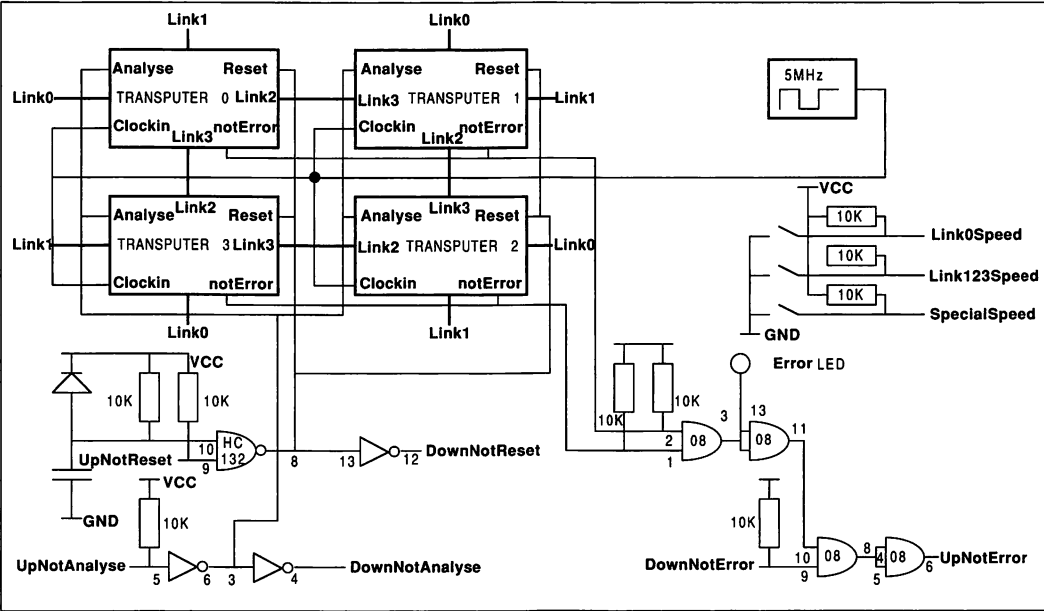
Figure 3.5 Logic shared by the four transputers on IMS B003

3.1.2   Logic used by all the transputers

The logic shared by all the transputers is shown in figure 3.5.

Reset etc

The evaluation boards share a common system control architecture. The aim of the system control functions is that it should be possible to control an arbitrarily large system built with the boards. The control implies the ability to reset the system, to note that an error has occurred in the system, and to analyse the error. Signals are provided for this purpose in the **Up** and **Down** sockets on the edge connector.

Up and **Down** sockets of evaluation boards are connected in a daisy chain as shown. The board at the top of the chain is controlled by a **Subsystem** socket on another evaluation board. The **Subsystem** socket has the same signals as the **Up** and **Down** sockets, but the **Subsystem** signals can be controlled by software running on the board with the **Subsystem** socket.

The **Reset** and **Analyse** signals flow in the direction of the arrows, the **Error** signal flows in the reverse direction from **Down** to **Up**, and indicates that an error has occurred on this board or on a board further down from this board.

All the B003's transputers are reset on power ON. A single Error LED (yellow) lights if an error has occurred on this board.

Coding switch

The coding switch sets the Link speed signals for all the transputers. Separate controls are provided for Links 0 and Links 123, which are independently set to 10 Mbits/s or 20 Mbits/sec.

Clock

The board uses a single 5 MHz clock oscillator, which is shared by all the transputers.

Systems

4   Designs and applications for the IMS C004

4.1   Introduction

The IMS C004 is a 32-way crossbar switch that supports the INMOS link protocol. This article describes its functionality, discusses how it may be used as a design element to provide larger crossbar switches, and how it may be applied to configure large transputer networks. It also suggests how it can be used as a general purpose communication engine, and gives an occam description of a message routing exchange.

It includes a concise description of the IMS C004's functionality using Hoare's CSP notation as well as a CSP description of the message routing exchange.
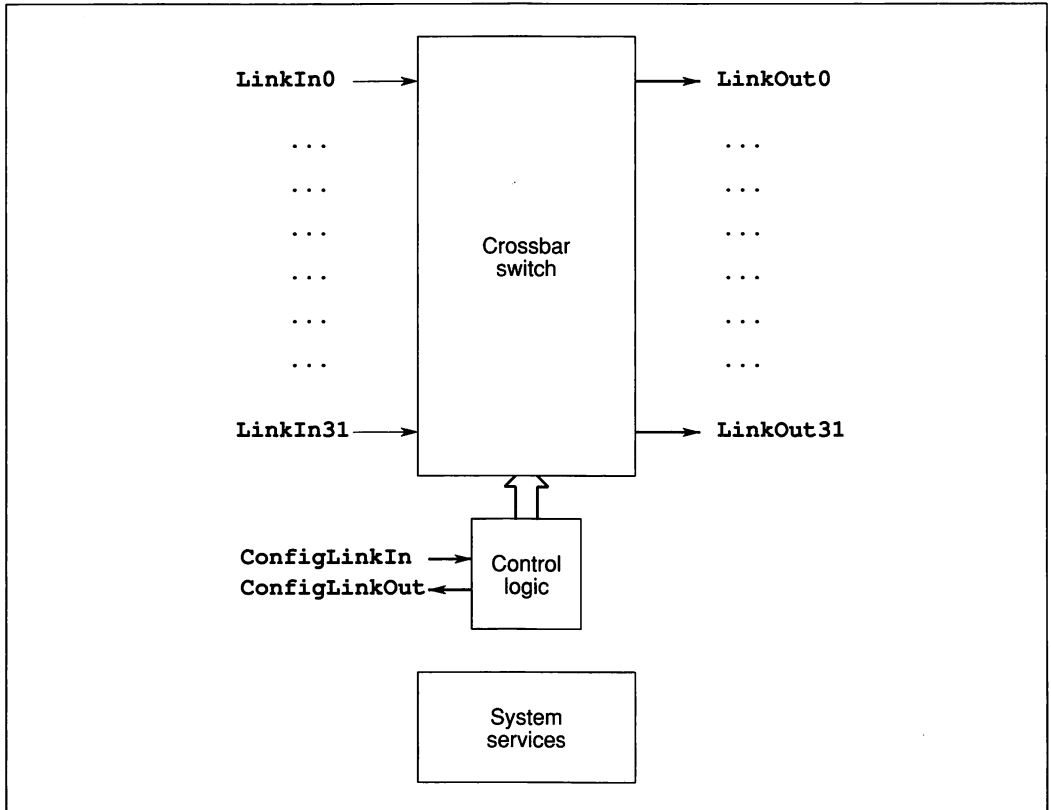
Figure 4.1 IMS C004 block diagram

4.2   IMS C004 programmable link switch

The INMOS communication link is a new standard for system interconnection. It uses the capabilities of VLSI to offer simple, easy-to-use and cheap interconnections for computer systems. The serial link is a fundamental component of, and was developed as part of, the INMOS transputer architecture. The transputer is a single VLSI device with memory, processor and communications links for direct connection to other transputers. It is a programmable component which enables systems to be constructed from a collection of transputers that operate concurrently and communicate through links.

The IMS C004 programmable link switch provides a full crossbar switch between 32 link inputs and 32 link outputs. It will switch links running at standard transputer speeds (10 and 20 Mbits/sec). It introduces a 1.6 to 2 bit time delay on the signal.

The link switch can be cascaded to any depth without loss of signal integrity and it can be used to construct reconfigurable networks of arbitrary size.

The IMS C004 is programmed via a separate serial link called the configuration link.

4.2.1   The INMOS serial link interface

```
              LinkOut  ─────────▶
              LinkIn   ◀─────────
              ClockIn  ◀───────── 5MHz
```
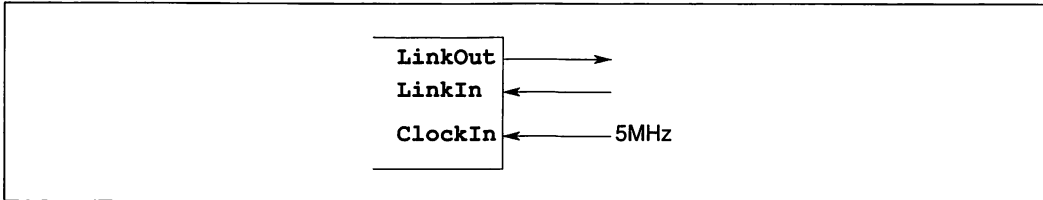
Figure 4.2 Standard clock input

INMOS serial links are standard across all products in the transputer product range. All transputers will support a standard communications frequency of 10 Mbits/sec, regardless of processor performance. Thus transputers of different performance can be connected directly and future transputer systems will be able to communicate directly with those of today. Each link consists of a serial input and a serial output, both of which are used to carry data and link control information.

A message is transmitted as a sequence of bytes. After transmitting a data byte, the sender waits until an acknowledge has been received, signifying that the receiver is ready to receive another byte. The receiver can transmit an acknowledge as soon as it starts to receive a data byte, so that transmission can be continuous. This protocol provides handshaken communication of each byte of data, ensuring that slow and fast transputers communicate reliably. When there is no activity on the links they remain at logic 0, **GND** potential.

A 5 MHz input clock is used, from which internal timings are generated. Link communication is not sensitive to clock phase. Thus communication can be achieved between independently clocked systems, provided that the communications frequency is within the specified tolerance.

```
Data      0   1   2   3   4   5   6   7
        ┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
        │ 1 │ 1 │   │   │   │   │   │   │   │ 1 │
        └───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘

Acknowledge
        ┌───┬───┐
        │ 1 │ 0 │
        └───┴───┘
```
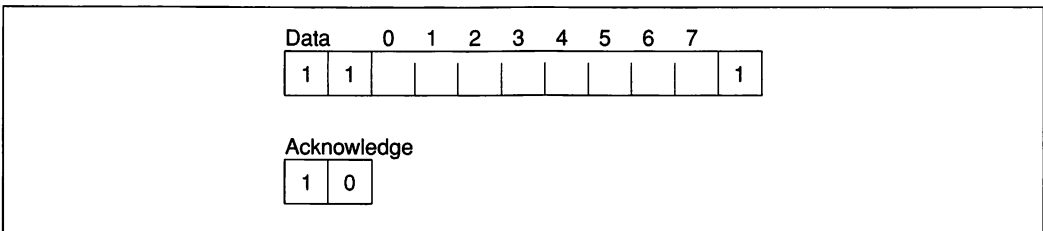
Figure 4.3 Link protocol

4.2.2   Switch implementation

The IMS C004 is internally organised as a set of thirty two 32-to-1 multiplexers. Each multiplexer has associated with it a six bit latch, five bits of which select one input as the source of data for the corresponding output. The sixth bit is used to connect and disconnect the output. These latches can be read and written by messages sent on the configuration link via **ConfigLinkIn** and **ConfigLinkOut**.

The output of each multiplexer is synchronised with an internal high speed clock and regenerated at the output pad. This synchronisation introduces, on average, a 1.75 bit time delay on the signal. As the signal is

not electrically degraded in passing through the switch, it is possible to form links through an arbitrary number of link switches.

Each input and output is identified by a number in the range 0 to 31. A configuration message consisting of one, two or three bytes is transmitted on the configuration link. The configuration messages sent to the switch on this link are shown in the table.

Configuration message	Function
[0] [input] [output]	Connects **input** to **output**.
[1] [link1] [link2]	Connects **link1** to **link2** by connecting the input of **link1** to the output of **link2** and the input of **link2** to the output of **link1**.
[2] [output]	Enquires which input the **output** is connected to. The IMS C004 responds with the input. The most signifigant bit of this byte indicates whether the output is connected (bit set high) or disconnected (bit set low).
[3]	This command byte must be sent at the end of every configuration sequence which sets up a connection. The IMS C004 is then ready to accept data on the connected inputs.
[4]	Resets the switch. All outputs are disconnected and held low. This also happens when **Reset** is applied to the IMS C004.
[5] [output]	Output **output** is disconnected and held low.
[6] [link1] [link2]	Disconnects the output of **link1** and the output of **link2**.

4.2.3   Functionality of the IMS C004

This section gives a textual description of the functionality of the IMS C004. For a more formal description refer to section 4.7.

As detailed in section 4.2.2, there are seven commands that are used to set up the IMS C004. (N.B. In first revision of silicon, the two disconnect commands were not included.) These will be referred to in this document as

ct.reset	(BYTE 4)
ct.input.output	(BYTE 0)
ct.link	(BYTE 1)
ct.enquire	(BYTE 2)
ct.disconnect.output	(BYTE 5)
ct.disconnect.link	(BYTE 6)
ct.setup	(BYTE 3)

These commands are sent to the IMS C004 via the configuration link (**ConfigLinkIn**, **ConfigLinkOut**). These single byte commands may be followed by output identifiers, input identifiers or link identifiers as explained below, all of which should be in the range BYTE 0 .. BYTE 31.

After power on reset, the single byte command *ct.reset* should be executed. This ensures that all inputs are disabled (i.e. cannot receive data) and all outputs are inactive (i.e. are not connected to any input).

The *ct.enquire* byte should be followed by an output identifier. The IMS C004 will then return, via the configuration link, an input identifier which represents the input to which that output is connected. This will be independent of whether or not that output is active. The most significant bit (bit 7) is set to 1 if the output is active. (N.B. In first revision of silicon this was not implemented.) Hence after a *ct.reset* command it is possible to find out to which input an output has been connected prior to the command. After a power on reset the input identifier returned after a *ct.enquire* command will be arbitrary.

The *ct.input.output* byte should be followed by an input identifier and an output identifier. This command enables the specified input, connects the specified output to that input and activates that output.

The *ct.link* byte should precede two link identifiers. This command is equivalent to two *ct.input.output* commands in which the identifiers are reversed; i.e.

ct.link link1 link2   =   *ct.input.output* link1 link2; *ct.input.output* link2 link1

The *ct.disconnect.output* byte should be followed by an output identifier. This command makes the specified output inactive.

The *ct.disconnect.link* byte should precede two link identifiers. This command is equivalent to two consecutive *ct.disconnect.output* commands; i.e.

ct.disconnect.link link1 link2   =   *ct.disconnect.output* link1; *ct.disconnect.output* link2

The *ct.setup* command is a single byte command that should be sent to the IMS C004 prior to using data links that have been redirected by the setup commands (*ct.input.output* or *ct.link*) to ensure that the IMS C004 has had enough time to be programmed correctly.
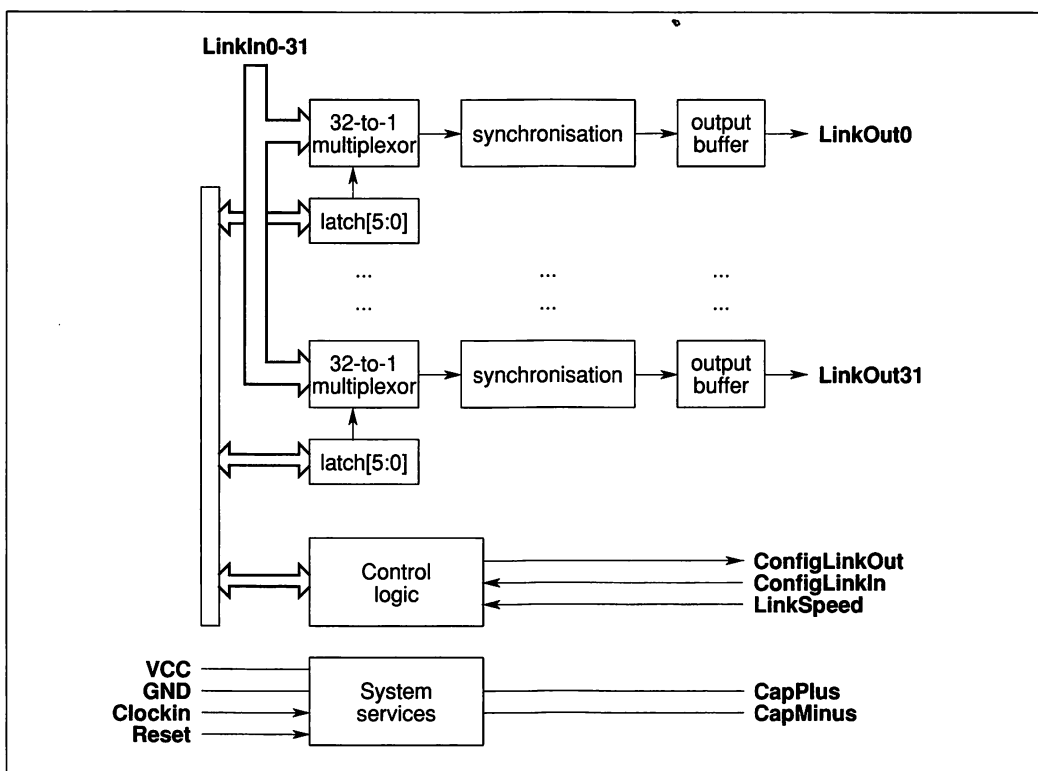
Figure 4.4 IMS C004 implementation

4.3   Versatility of the IMS C004

Since IMS C004's are digital devices that effectively regenerate received data for transmission, they can be used as elements of larger switching networks without any signal degradation occuring when a link path is routed through several elements. The only drawback is that each IMS C004 can introduce a delay of up to 2 bits, and since each byte transfer requires a data and acknowledge packet to comply with the link protocol, the communication bandwidth is reduced by each IMS C004.

The IMS C004 is a 32-way crossbar switch. This doesn't however restrict a designer to using a crossbar of this size. Large crossbars can be designed from smaller crossbar elements. This section introduces two possible design methods to achieve this, and describes how these methods can be used for cascading IMS C004s.

4.3.1   A small increase in crossbar capacity

If a crossbar element of size **M** is available (**M** = 32 for an IMS C004) and a design requires a slightly larger crossbar, this can be achieved using three crossbars to produce a single crossbar of greater capacity. Figure 4.8 shows a special case where three identical crossbars (size **M**) are combined to produce a 50% larger crossbar (size 3**M**/2). The following text explains why this arrangement achieves the objective.

Assume that an **N**-way crossbar is required. That is, a circuit that can connect **N** inputs to **N** outputs in any permutation.

A trivial way of doing this is shown in figure 4.5. It is immediately obvious that this design has not achieved anything, since two **N**-way crossbars have been merged to derive a single **N**-way crossbar. Nevertheless, it is easy to see that the required circuit has been produced.
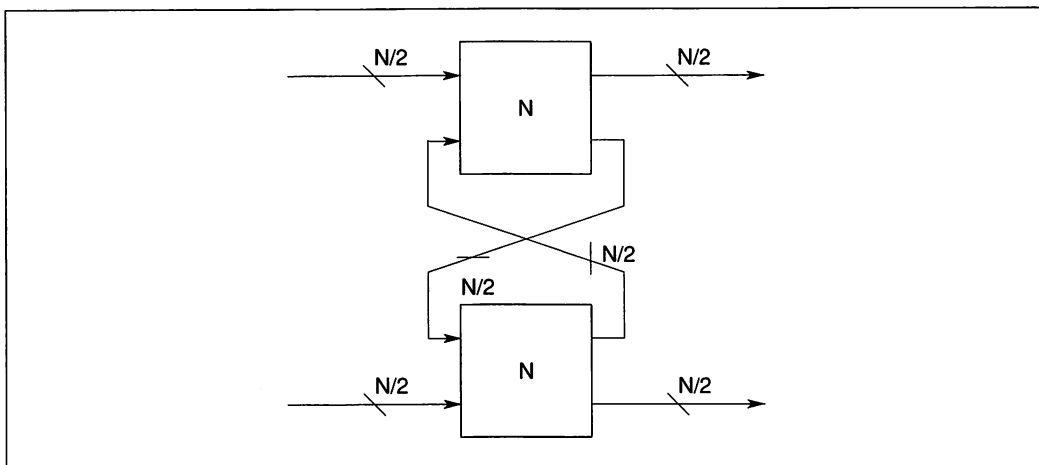
Figure 4.5

Another design that achieves our objective is shown in figure 4.6. Provided that we are happy with the design of figure 4.5, it is not very difficult to convince ourselves that this new design will also satisfy the requirement that any input can be connected to any output. If any input needs to be connected to either output 0 or output 1, then it must be routed via the 2-way crossbar. This still is not a particularly useful design, since there is a great deal of expense in producing a crossbar only one dimension larger than the two needed to implement it.
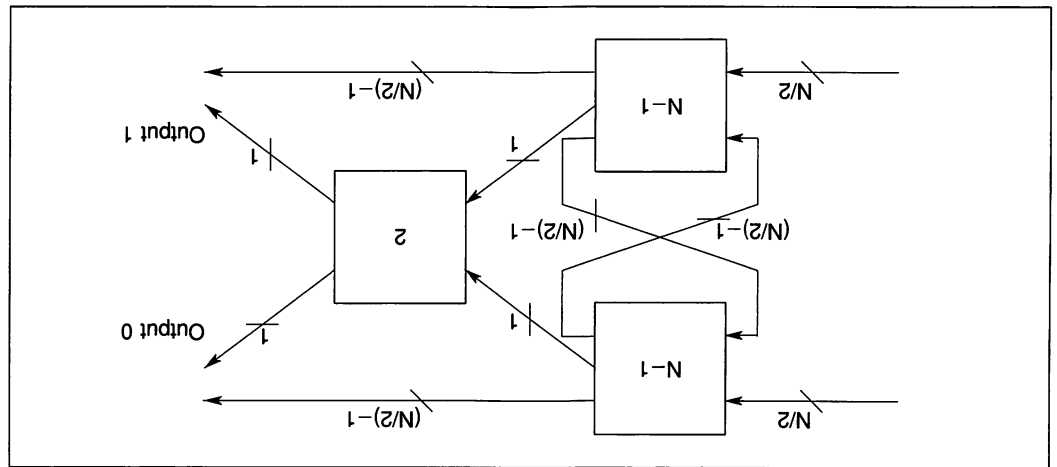
Figure 4.6

However, the concept is important because there is no reason why we cannot increase the size of the smaller crossbar, hence reducing the size of the larger ones to achieve the same result. Figure 4.7 shows the generalised design structure for combining three crossbars in this way to produce a larger crossbar. Now, if all three crossbars are of size M they combine to derive a crossbar of size $3M/2$ (figure 4.8).

Three IMS C004s can therefore be used to implement a 48-way crossbar.
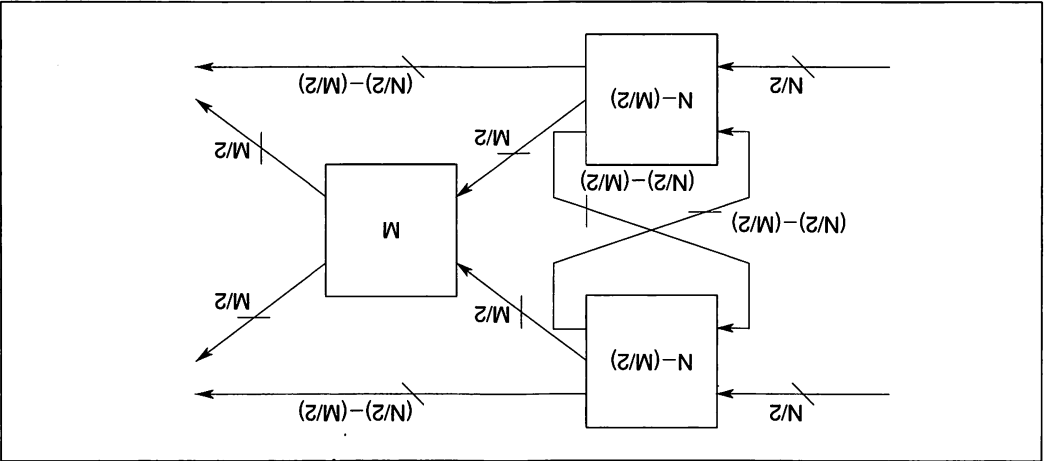
Figure 4.7
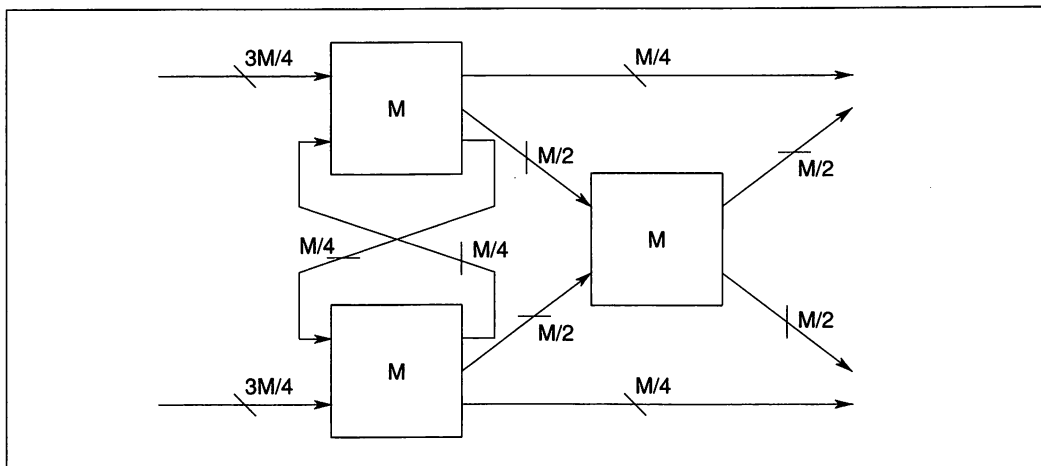
Figure 4.8
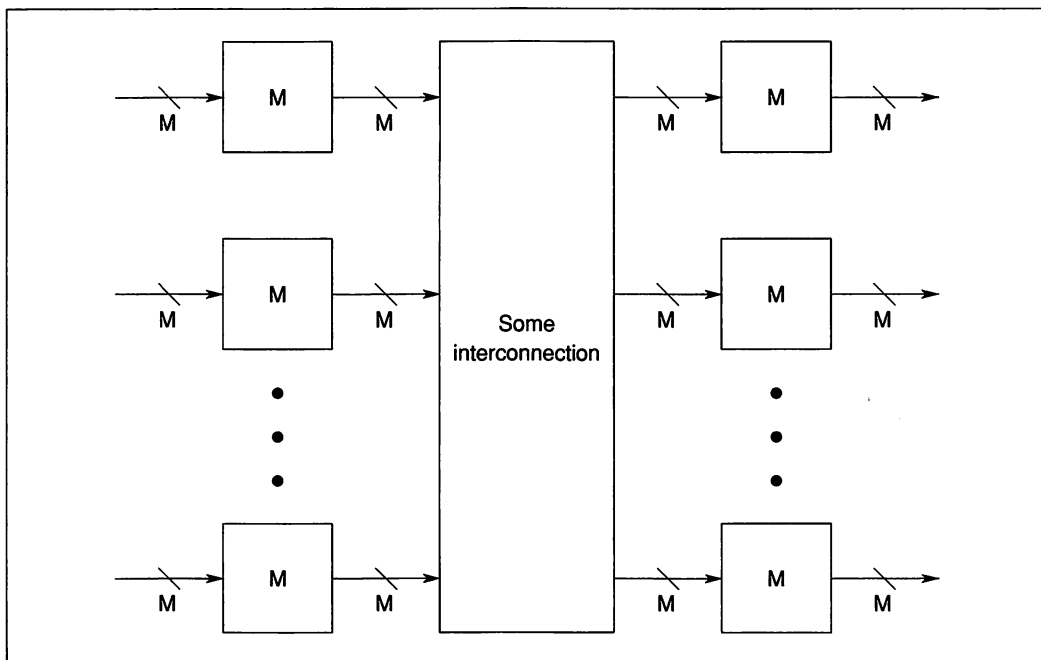
Figure 4.9 Large crossbar design using smaller crossbar elements

4.3.2   A large increase in crossbar capacity

A large crossbar can be derived from smaller crossbar elements (**M**-way) as shown in figure 4.9. A first attempt at defining the unknown block might be a simple interconnection as shown in figure 4.10. But an obvious requirement for figure 4.9 is that there should be at least **M** paths between any input crossbar and output crossbar, which figure 4.10 does not satisfy.
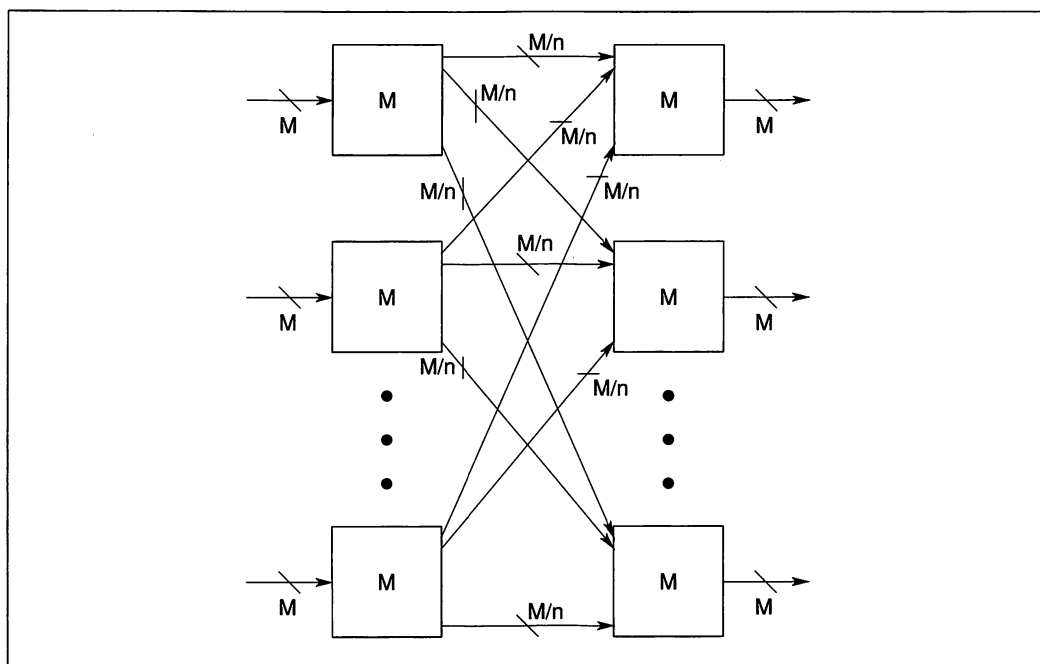
Figure 4.10 A first attempt

An arrangement which does satisfy this requirement is shown in figure 4.11. This uses **3n** elements of size **M** to implement an **nM**-way crossbar where **n** $\leq$ **M**. A crossbar switch with **M** inputs and **M** outputs can be used to design a crossbar with up to **M**^{2} inputs and **M**^{2} outputs. Note that it also has the property that each input to output connection will always be routed through three of the smaller elements.

But note that since we cannot have a fraction of a link, this description uses integer arithmetic. In general, therefore, it is possible to design a crossbar of size **n(M** $-$ **M** mod **n)**.

Using this assertion here are some examples for a C004 (where **M**=32):

n	C004s	Size of crossbar
2	6	64
3	9	90
4	12	128
5	15	150
.	.	.
.	.	.
32	96	1024
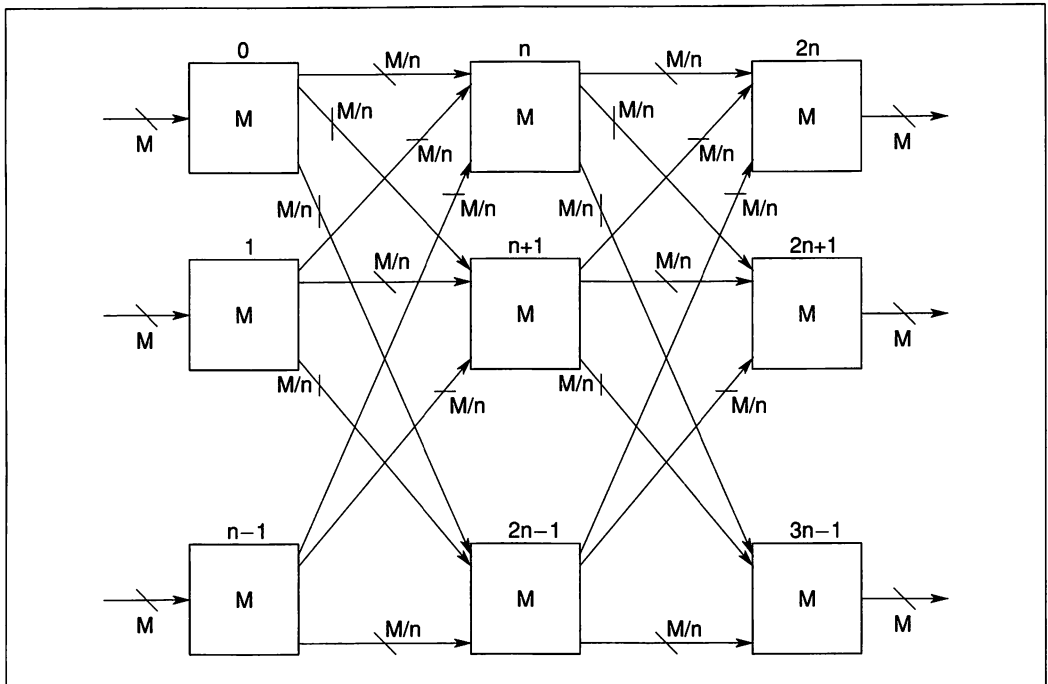
Figure 4.11 An nM-way crossbar design for a fixed delay

4.3.3   Design example for cascading IMS C004s

From section 4.3.1, it can be seen that three IMS C004's can be cascaded to derive a 48-way crossbar, and from section 4.3.2 that $3n$ IMS C004's can be used to achieve a crossbar of size $n(32 - 32 \bmod n)$ for $n \leq 32$.

Sometimes a choice must be made between the two design techniques. For example if two 45-way crossbars are required, then the first design could be implemented using six IMS C004's (three IMS C004's for each crossbar). Alternatively, two 45-way crossbars are a subset of a single 90-way crossbar (which has the bonus of extra flexibility), and this can be implemented using nine IMS C004's in the second design. If such a choice is to be made then the following properties should be considered. The first design will route each link path through 1,2 or 3 IMS C004s, whereas the other will always route through three IMS C004's. The average link delay of the first will therefore be smaller, which will usually be preferable, but a fixed link delay might be more desirable. The software support for setting up the second cascade is simpler because the design is more uniform and the crossbar is more flexible. Finally the first design will use fewer IMS C004's.

4.4   Using the IMS C004 to configure transputer networks
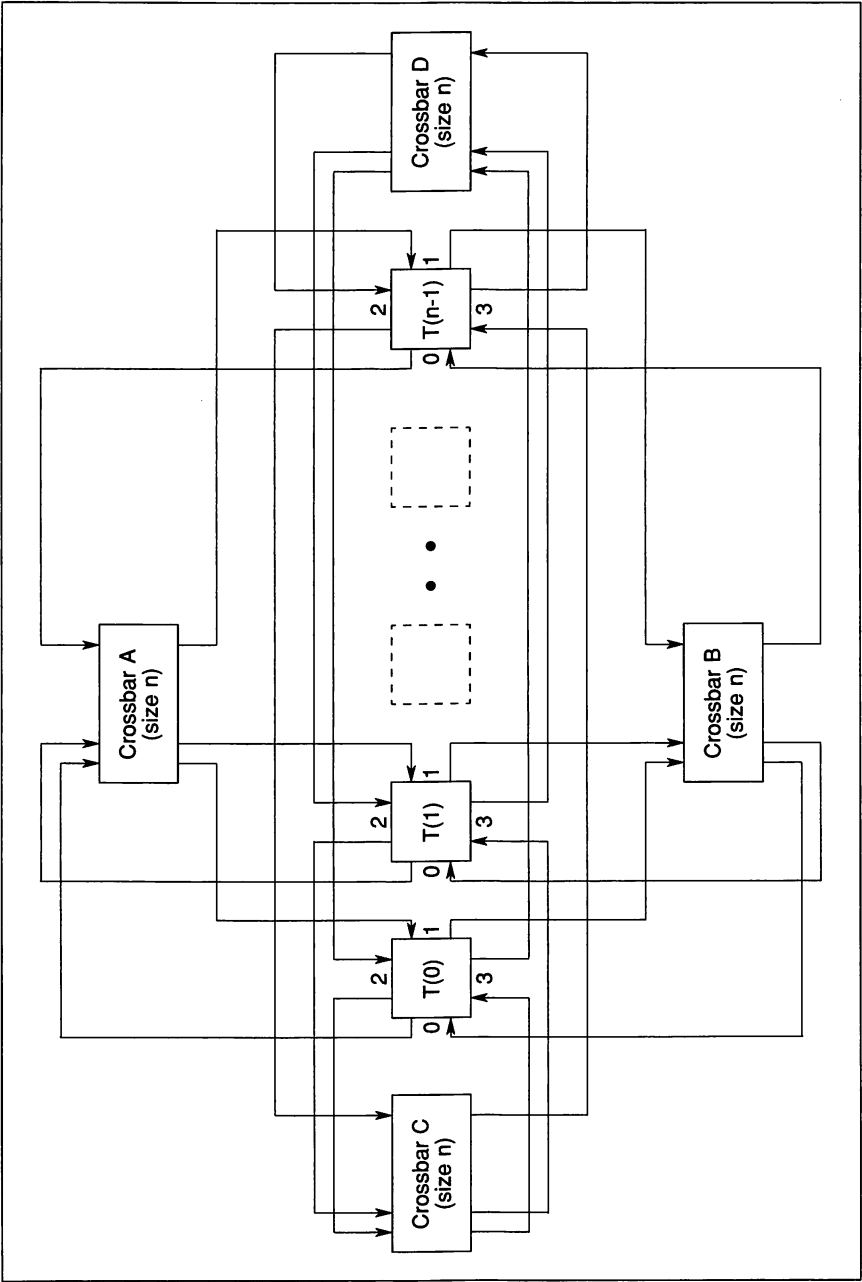
Figure 4.12 Complete connectivity of a transputer network using four crossbars

4.4.1   Complete connectivity of a transputer network using four crossbars

The design suggested in this section makes use of the property that all four transputer links are identical. This means that as far as the configuration software is concerned, it doesn't care on which link a hard channel is placed, provided that each is connected to the transputer specified by that software. Because of this we can choose any link numbering scheme when trying to configure a network with crossbars.

It is always possible to set a network of transputers to any configuration using just four crossbars. The size of the crossbars should be at least as great as the number of transputers in the network. For example, a 32 node network can be configured using four IMS C004's, and a 48 node network can be configured using twelve (making use of an IMS C004 cascade arranged as shown in figure 4.8). Although a complete proof of this statement is outside the scope of this text, we will show how this can be achieved for configurations that contain a Hamitonian Cycle (i.e. a route through the network that visits every node once only). This method will be applicable to most interesting configurations. The hardware arrangement is as shown in figure 4.4. Note that crossbar **A** connects *link 0* outputs to *link 1* inputs, crossbar **B** connects *link 1* outputs to *link 0* inputs, and crossbars **C** and **D** similarly connect links 2 and 3.

Firstly, find a Hamiltonian Cycle (if one exists) through the network and choose a *link 0* to *link 1* connection between all transputers. Since any *link 0* can be connected to any *link 1* by crossbars **A** and **B** this cycle can be configured.

Now each transputer has just two links left to connect. Again since these links are identical, we do not care which links we choose when connecting our configuration.

If, for example, transputer **p** is to be connected to transputer **q** (figure 4.13) and so far no other connections have been made, a *link 2* to *link 3* connection can be made in one of two ways. Having made this connection (figure 4.14), transputer **q** *link 3* can be connected to *link 2* of any other transputer in the network (including **p**). If another link between **p** and **q** is required, these transputers will be completely connected (i.e. there cannot be other connections to them) and so the next link to be connected will be between two transputers with both *link 2* and *link 3* unconnected.
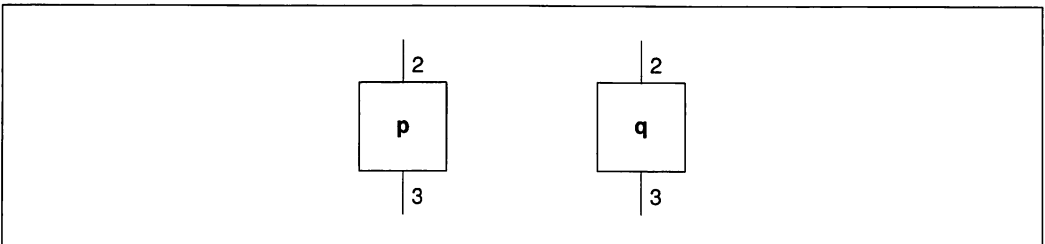
Figure 4.13
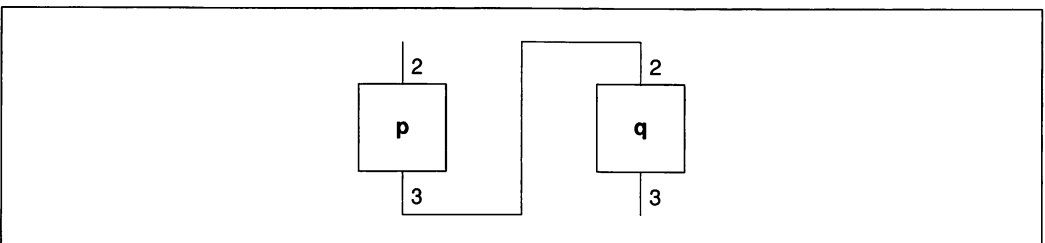
Figure 4.14

Assume now that **q** is connected to transputer **r** (figure 4.15). *link 3* of transputer **r** can be connected to *link 2* of any other transputer in the network with the exception of transputer **q**. But since *link 2* and *link 3* of **q** have already been connected, it will not be required to connect another link to it in a four link configuration. If a link between **r** and **p** is required, we again have a completely connected group.
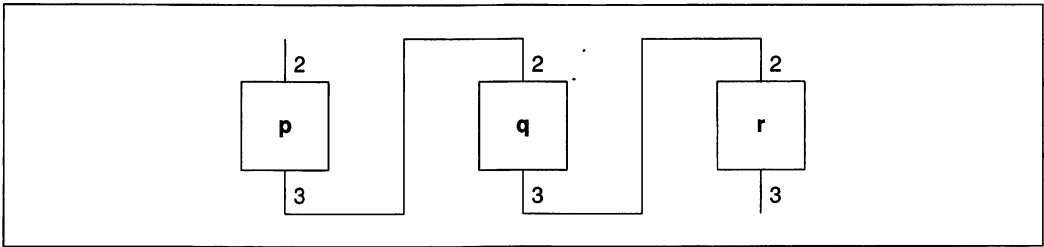
Figure 4.15

Hence, by induction, it is always possible to arrange that all *links 2* are connected to *links 3* and vice-versa. This can be achieved using crossbars **C** and **D** in figure 4.4.

4.4.2   Complete connectivity of a transputer network using two crossbars

In the previous section, advantage was taken of the fact that all transputer links are identical. It will often also be true that all transputers in the network are identical. If this is the case then the Hamiltonian Cycle (if it exists) can be a fixed pipeline through the network. This means that the *link 0* to *link 1* connections can be hardwired and all possible configurations can be obtained by connecting *link 2* to *link 3* using two crossbars as described above. A network of **N** transputers could then be configured using just two **N**-way crossbars. This arrangement is shown in figure 4.4.2.

For example 32 transputers can be completely configured using just two IMS C004s.
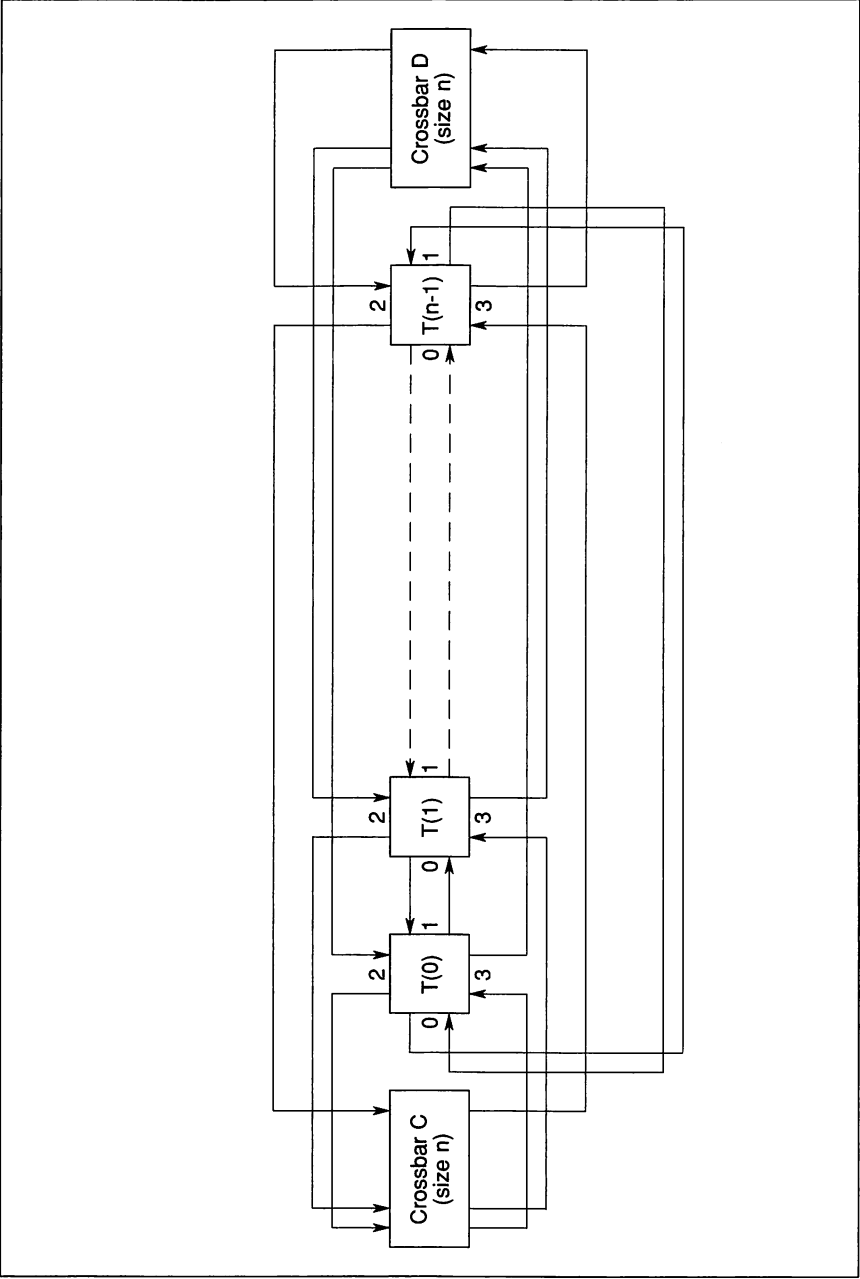
Figure 4.16 Complete connectivity of a transputer network using two crossbars

4.5   Using the IMS C004 as a general purpose communication crossbar

The use of the IMS C004 is not restricted to computer configuration applications. The ability to change the switch setting dynamically enables it to be used as a general purpose message router. This may of course also find applications in computing with the emergence of the new generation of supercomputers, but a more widespread use may be found commercially as a communication exchange.

This section considers one way in which an exchange might be implemented. A suitable protocol for this example is shown using Hoare's CSP notation [CAR Hoare: Communicating Sequential Processes] in section 4.8. A possible occam implementation is included below for users unfamiliar with CSP. There is no reason why this exchange should not be expanded with a larger crossbar, making use of the design techniques of section 4.3.

A message into the exchange must be preceded by a destination token. When this message is routed through the exchange, the destination token is replaced with a source token so that the receiver knows where the message has come from. The *input.output* processes of figure 4.17 and the *controller* processes could be implemented easily with INMOS IMS T212 transputers, and the link protocol for establishing communication with these devices can be interfaced with INMOS link adaptors. In this configuration two channels are placed on each IMS C004 link in opposite directions.
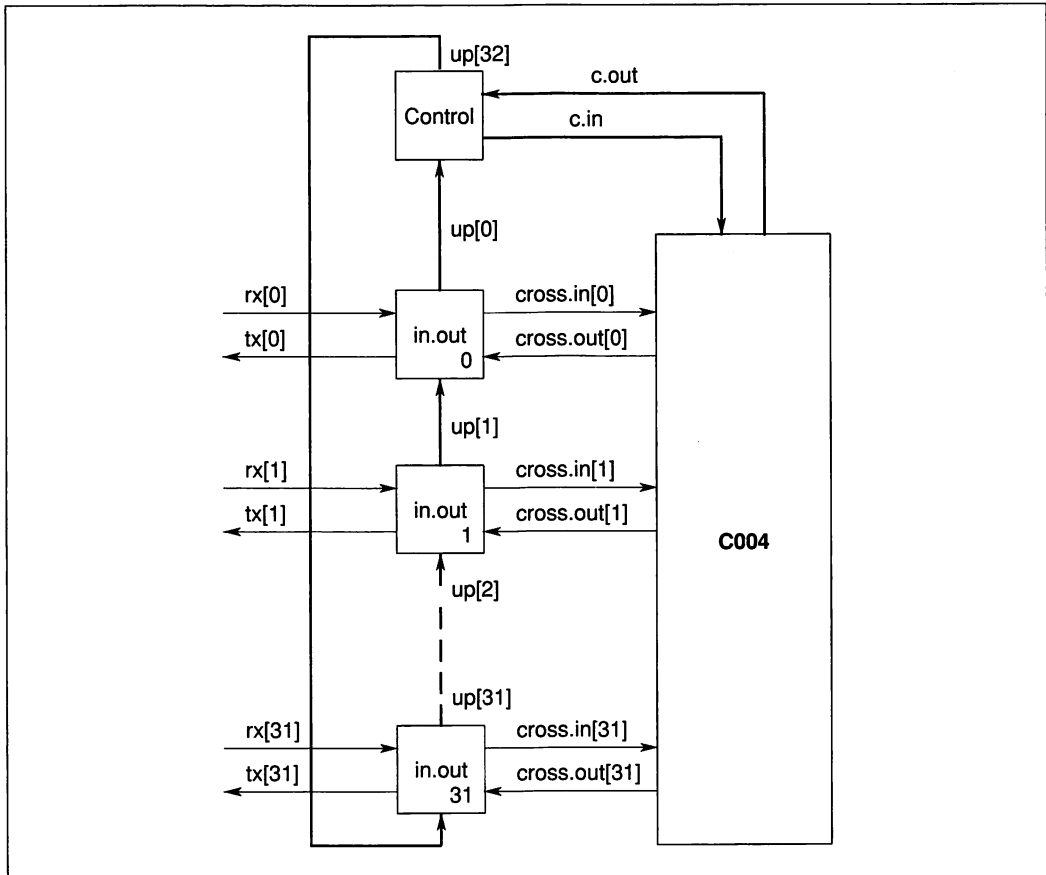
Figure 4.17

4.5.1   occam **implementation of a 32 stage bidirectional exchange**

This section provides some occam code that could be used to implement the exchange described in section 4.8. Its main purpose within the context of this document is to give an alternative way of describing the example for the reader who is unfamiliar with CSP. For this reason, declarations have been omitted except where confusion might arise without (figure 4.17).

```
PLACED PAR
  PROCESSOR no.of.nodes T2
    controller (c.in, c.out,
                up[0],
                up[no.of.nodes])
  PLACED PAR i = 0 FOR no.of.nodes
    PROCESSOR i T2
      input.output (BYTE i,
                    rx[i], tx[i],
                    up[i],
                    up[i+1],
                    cross.in[i], cross.out[i])
```

Notes

1 Link placement statements have been omitted, but a convention has been adopted that two channels placed on the same bidirectional link are paired together on the same line. All channel parameters are hard channels.

2 Constant byte tokens are prefixed by *ct.* for IMS C004 tokens and *et.* for exchange tokens.

3 Section 4.2 recommends that a *ct.setup* token is sent to the configuration link of the IMS C004 after a *ct.link* command. The reason for this is to give the IMS C004 enough time to make the connection. In this application there will be a substantial delay before that connection is used by an *input.output* process and so this precaution is not necessary.

Controller

The code for this process should be loaded onto the transputer that talks to the IMS C004 via its configuration link. It receives a token from hard channel **up.in** and, depending on the value of that token, takes one of three paths before repeating.

```
PROC controller (CHAN c.in, c.out,
                       up.in,
                       up.out)
  WHILE TRUE
    SEQ
      up.in? token
      IF
        token = et.ack
          --  consume rest of acknowledge packet since
          --  it has done its job
          up.in? any.byte; any.byte
        token = et.req
          ...  deal with request                          -- (i)
        token = et.rel
          ...  setup link or send new request             -- (ii)
  :
```

i. deal with request

This firstly receives the rest of the request packet. It then finds out which nodes are currently connected to the two that want to talk to each other and sends a release packet to inform the relevant nodes that a new link is about to be set up.

```
{{{ deal with request
SEQ
  up.in?  source; dest
  c.in!   ct.enquire; source
  c.out? current.source.conn    -- address of node currently
                                -- connected to source
  set.to.nil.if.inactive (current.source.conn)            -- (iii)
  c.in!   ct.enquire; dest
  c.out? current.dest.conn      -- address of node currently
                                -- connected to dest
  set.to.nil.if.inactive (current.dest.conn)              -- (iii)
  up.out! et.rel; current.source.conn;
          current.dest.conn; source; dest
}}}
```

ii. setup link or send new request

This firstly receives the rest of the release packet. It then proceeds to find out what is currently connected to the two that want to communicate. If the same as before (i.e. when this was done before sending the release packet) then the previous connections are disconnected, the new link is set up, and an acknowledge packet is transmitted. Otherwise a new release packet is sent.

```
{{{ setup link or send new request
SEQ
  up.in?  last.source.conn; last.dest.conn; source; dest
  c.in!  ct.enquire; source
  c.out? current.source.conn
  set.to.nil.if.inactive (current.source.conn)            -- (iii)
  c.in!   ct.enquire; dest
  c.out? current.dest.conn
  set.to.nil.if.inactive (current.dest.conn)              -- (iii)
  IF
    (last.source.conn = current.source.conn) AND
                  (last.dest.conn = current.dest.conn)
      -- IMS~C004 setup has not affected these node connections
      -- since the release packet was transmitted
      SEQ
        --   disconnect current.source.conn and source
        IF
          current.source.conn = byte.nil
            SKIP
          TRUE
            -- disable current connection to source
            c.in! ct.disconnect.link; current.source.conn; source
        --   disconnect current.dest.conn and dest
        IF
          current.dest.conn = byte.nil
            SKIP
          TRUE
            -- disable current connection to dest
            c.in! ct.disconnect.link; current.dest.conn; dest
        c.in!   ct.link; source; dest
        up.out! et.ack;  source; dest
    TRUE
      SEQ
        -- transmit a new release packet
        up.out! et.rel; current.source.conn;
                current.dest.conn; source; dest
}}}
```

iii. set.to.nil.if.inactive

If bit 7 of the parameter **output.conn** is 1, then the connection is inactive and the byte is set to address **nil**. Otherwise it is unchanged. This could not be expressed in detail in the CSP description.

```
PROC set.to.nil.if.inactive (BYTE output.conn)
  IF
    (output.conn BITAND (BYTE #80)) = (BYTE #80)
      output.conn := byte.nil
    TRUE
      SKIP
:
```

Input.Output

The code for this process should be loaded onto all the other transputers. The state is initially inactive. If a message is received from the IMS C004 on **switch.in** then it is passed on via **data.out**. If a command packet is received on **up.in** then it is dealt with as described in (iv). If a message is received on **data.in** then it is dealt with as described in (v). This repeats indefinitely. Note that a priority is given to the three input sequences. This could not be expressed in CSP.

```
PROC input.output (VAL BYTE i,
                   CHAN data.in, data.out,
                        up.out,
                        up.in,
                        switch.out,switch.in)
  SEQ
    state := inactive
    d := byte.nil
    [max.mess]BYTE rx.mess:
    [max.mess]BYTE tx.mess:
    WHILE TRUE
      PRI ALT
        switch.in? source; tx.length; [tx.mess FROM 0 FOR tx.length]
          data.out! source; tx.length; [tx.mess FROM 0 FOR tx.length]
        up.in? token
          ...  deal with command packet                          -- (iv)
        ((state = active) OR (state=inactive)) &
         data.in? dest; rx.mess; [rx.mess FROM 0 FOR rx.length]
          ...  deal with message transfer
:                                                                 -- (v)
```

iv. deal with command packet

If a release token has been received, the rest of the release packet is received and passed on to the next node. Now if the state is active and either **dest**, **addr1** or **addr2** are the same as the local identifier (**i**), then the state is set to inactive. Note that this is not necessary if the link that has been requested already exists, which may occur if the other end of the link has made the request prior to the existing setup.

If a request token has been received, the rest of the packet is received and passed on since this will only be analysed by the controller.

If an acknowledge token has been received, the rest of the packet is received and passed on. If the destination address is local (**dest** = **i**) then a new link path has been set up for this node and it becomes active. If the source address is local (**source** = **i**) then the request that was previously sent has now been acknowledged and the stored message can be sent to its destination via the IMS C004.

```
{{{ deal with command packet
IF
  token = et.rel
    SEQ
      up.in?  addr1; addr2; source; dest
      -- pass release packet on to next node in daisy chain
      up.out! et.rel; addr1; addr2; source; dest
      IF
        (state = active) AND
        ((((addr1 = i) OR (addr2 = i)) OR (dest = i)) AND
        (NOT ((source=d) AND (dest=i))))
          -- another node has requested a link to this node or its
          -- connected node is to be connected to another node
          state := inactive
        TRUE
          SKIP

  (token = et.req) OR (token = et.ack)
    SEQ
      up.in? source; dest
      -- pass request or acknowledge packet on to
      -- next node in daisy chain
      up.out! token; source; dest
      IF
        token = et.req
          SKIP
        token = et.ack
          IF
            (state = inactive) AND (dest = i)
              -- a link has been set up with
              -- another node
              SEQ
                state := active
                d := source
            (state = pending) AND (source = i)
              -- the link that was previously requested
              -- has now been set up
              SEQ
                switch.out! i; rx.length;
                            [rx.mess FROM 0 FOR rx.length]
                state := active
                d := dest
            TRUE
              SKIP
}}}
```

v. deal with message transfer

A message has been received with an associated destination. If the state of the process is active and the destination is that already set up (**dest** = **d**) then the message can be immediately routed through the IMS C004. Otherwise a request is sent to the controller to set up a new link path and the state is set to pending.

```
{{{ deal with message transfer
SEQ
  IF
    (state = active) AND (dest = d)
      -- the destination requested by the message
      -- received is the one that is currently
      -- connected by the IMS~C004
      switch.out! i; rx.length;
                   [rx.mess FROM 0 FOR rx.length]
    (state = active) OR (state = inactive)
      -- a new link needs to be requested
      SEQ
        up.out! et.req; i; dest
        state := pending
}}}
```

4.5.2   Comment on exchange activity

While a message transfer is occurring, two **input.output** processes of the bidirectional exchange will become busy and will not be able to pass information to the controller. For this reason messages should be kept short and long messages should be broken into short ones. In the case, for example, when all routes are active in transferring data between fixed destinations and sources, there need not be any communication to the controller until a particular source decides it wants to talk to another destination. Therefore for the exchange to operate efficiently each **input.output** process would be expected to be predominantly in the **active** state.

4.6   Conclusions

A single IMS C004 can be used alone as a 32 x 32 crossbar supporting INMOS link protocol. Alternatively, since it is a digital device, a number of IMS C004's can be used to construct a larger crossbar without any other hardware. Since it introduces a small real time communication delay, the data transmission rate will be reduced when cascading more than one IMS C004.

With careful design and suitable software support, a small number of IMS C004's can be used to completely connect any configuration of a large network of transputers without any loss of generality.

Since it can be dynamically programmed, its applications can be extended to systems that might not use transputers. The INMOS link adaptor enables any parallel bus users to take advantage of the flexibility of the device. The design of a message routing exchange is fairly straightforward.

4.7   CSP description of IMS C004

For completeness a concise description of the IMS C004 is given using CSP [CAR Hoare: Communicating Sequential Processes].

N.B. Protocol tokens are prefixed by *ct.* for external tokens and *cit.* for internal tokens.

c.in and *c.out* are the channels associated with the control link (or configuration link). *cross.in* and *cross.out* represent the link input and link output wires which are connected by the crossbar. The protocol tokens: *ct.input.output*, *ct.link*, *ct.enquire*, *ct.disconnect.output*, *ct:disconnect.link* and *ct.reset* correspond to bytes: 0, 1, 2, 3, 5 and 4 respectively.

$$IMS.C004 = C \parallel (\mathop{\parallel}_{i \in 0..31} IN_i) \parallel (\mathop{\parallel}_{j \in 0..31} OUT_j)$$

The process IN_i has two states. It is initially set to $IN_{i,\emptyset}$. As information is received from process C, the parameter *set* is modified. When *set* is not empty, the process may either receive a message packet (*mess*) from cross.in[i], in which case *mess* is sent to all OUT_j processes that are referenced by the elements of *set*, or the process may receive a token *op* from C. In the latter case, if *op* is *ct.reset*, the parameter *set* becomes empty and the process continues to behave like $IN_{i,\emptyset}$, but if the token is *cit.sub* or *cit.add*, it receives another token from C and either subtracts or adds this to *set* depending on whether *op* is *cit.sub* or *cit.add*. $IN_{i,\emptyset}$ can only receive information from C.

IN_i = $IN_{i,\emptyset}$

$IN_{i,\emptyset}$ =   setup.in$_i$? op → *case*   op = cit.reset   ⇒ $IN_{i,\emptyset}$
                                                op = cit.sub   ⇒   setup.in$_i$? any → $IN_{i,\emptyset}$
                                                op = cit.add   ⇒   setup.in$_i$? output → $IN_{i,\{output\}}$
                              esac

$IN_{i,set}$ =   setup.in$_i$? op → *case*   op = cit.reset   ⇒ $IN_{i,\emptyset}$
                                                op = cit.sub   ⇒   setup.in$_i$? output → $IN_{i,(set-\{output\})}$
                                                op = cit.add   ⇒   setup.in$_i$? output → $IN_{i,(set\cup\{output\})}$
                              esac

| cross.in[i]?   mess   → $\mathop{\parallel}_{j \in set}$ $c_{i,j}$!   mess   → $IN_{i,set}$

The process OUT_j has two states. It is initially set to $OUT.INACTIVE_{0,j}$. In the state $OUT.INACTIVE_{i,j}$ it can receive an input address *in* from C which if not *nil*-will set the state to $OUT.ACTIVE_{in,j}$, or it can receive a token from C on a separate channel to which it responds by returning the process state and the identifier of the IN_i process from which it has been set up to receive messages (*i*). In the state $OUT.ACTIVE_{i,j}$ the process may follow either of the paths described above or it may receive a message packet (*mess*) from IN_i which is then transmitted on *cross.out[j]*.

$OUT_j = OUT.INACTIVE_{0,j}$

$OUT.INACTIVE_{i,j} =$ setup.out$_j$? in → $(OUT.INACTIVE_{i,j} \not< in = nil \not> OUT.ACTIVE_{in,j})$

                       | enquire$_j$? any → answer$_j$! false, i → $OUT.INACTIVE_{i,j}$

$OUT.ACTIVE_{i,j} =$ setup.out$_j$? in → $(OUT.INACTIVE_{i,j} \not< in = nil \not> OUT.ACTIVE_{in,j})$

                       | enquire$_j$? any → answer$_j$! true, i → $OUT.ACTIVE_{i,j}$

                       | $c_{i,j}$? mess → cross.out[j]! mess → $OUT.ACTIVE_{i,j}$

The process C will receive an item *token* on the *c.in* channel. Depending on which command this is, the process branches to one of six different processes.

C =   c.in? token
        $\rightarrow$ *case*   token = ct.input.output $\Rightarrow$ SET.IN.OUT
                       token = ct.link $\Rightarrow$ SET.LINK
                       token = ct.enquire $\Rightarrow$ ENQUIRE
                       token = ct.disconnect.output $\Rightarrow$ DISCONNECT.OUTPUT
                       token = ct.disconnect.link $\Rightarrow$ DISCONNECT.LINK
                       token = ct.reset $\Rightarrow$ RESET_0
                       token = ct.setup $\Rightarrow$ C
       esac

SET.IN.OUT receives the *input* and *output* addresses to be connected. It then enquires as to which link input (*last.input*) was previously talking to OUT_{output} and sends a (*cit.sub, output*) packet to $\text{IN}_{last.input}$. It sets up the new connection by sending *input* to OUT_{output} and a (*cit.add, output*) packet to IN_{input}.

SET.IN.OUT =   c.in? input, output $\rightarrow$ enquire_{output}! any $\rightarrow$ answer_{output}? any; last.input
                  $\rightarrow$ $\text{setup.in}_{last.input}$! cit.sub, output $\rightarrow$ $\text{setup.out}_{output}$! input
                  $\rightarrow$ setup.in_{input}! cit.add, output $\rightarrow$ C

SET.LINK receives the *link1* and *link2* addresses to be connected. It then finds out which link input (*last.input*) was previously talking to OUT_{link1} and sends a (*cit.sub, link1*) packet to $\text{IN}_{last.input}$, and repeats this procedure for *link2*. It sets up the new connection by sending *link1* to OUT_{link2} and *link2* to OUT_{link1}, followed by a (*cit.add, link2*) packet to IN_{link1} and a (*cit.add, link1*) packet to IN_{link2}.

SET.LINK =   c.in? link1, link2
               $\rightarrow$ enquire_{link1}! any $\rightarrow$ answer_{link1}? any; last.input $\rightarrow$ $\text{setup.in}_{last.input}$! cit.sub, link1
               $\rightarrow$ enquire_{link2}! any $\rightarrow$ answer_{link2}? any; last.input $\rightarrow$ $\text{setup.in}_{last.input}$! cit.sub, link2
               $\rightarrow$ setup.out_{link1}! link2 $\rightarrow$ setup.out_{link2}! link1
               $\rightarrow$ setup.in_{link1}! cit.add, link2 $\rightarrow$ setup.in_{link2}! cit.add, link1 $\rightarrow$ C

ENQUIRE receives the link *output* for enquiry and sends an arbitrary token to the relevant OUT process (via enquire_{output}) which responds (via answer_{output}) with the appropriate link *input* that is assigned to this *output* and a boolean state to determine whether this connection is activated. (N.B. In the implementation this is encoded into the byte that also contains the input, i.e bit 7 contains state, bits 4 .. 0 contain input address). This is then transmitted on *c.out*.

ENQUIRE =   c.in? output $\rightarrow$ enquire_{output}! any $\rightarrow$ answer_{output}? status, input
               $\rightarrow$ c.out! status, input $\rightarrow$ C

DISCONNECT.OUTPUT receives the link *output* address to be disconnected. It determines which link *input* was previously connected to it, sends a (*cit.sub, output*) packet to IN_{input} and sends *nil* to OUT_{output}.

DISCONNECT.OUTPUT =   c.in? output $\rightarrow$ enquire_{output}! any $\rightarrow$ answer_{output}? any, input
                      $\rightarrow$ setup.in_{input}! cit.sub, output $\rightarrow$ $\text{setup.out}_{output}$! nil $\rightarrow$ C

DISCONNECT.LINK receives the link addresses (*link1* and *link2*) to be disconnected, and does the same as DISCONNECT.OUTPUT for each address.

DISCONNECT.LINK =   c.in? link1, link2 $\rightarrow$ enquire_{link1}! any $\rightarrow$ answer_{link1}? any, input
                  $\rightarrow$ setup.in_{input}! cit.sub, link1 $\rightarrow$ setup.out_{link1}! nil
                  $\rightarrow$ enquire_{link2}! any $\rightarrow$ answer_{link2}? any, input
                  $\rightarrow$ setup.in_{input}! cit.sub, link2 $\rightarrow$ setup.out_{link2}! nil $\rightarrow$ C

RESET sends *ct.reset* to all IN processes and *nil* to all OUT processes. All IN and OUT processes are therefore deactivated but the OUT processes preserve knowledge of their previous connection.

$RESET_{31}$ = setup.in$_{31}$! cit.reset $\rightarrow$ setup.out$_{31}$! nil $\rightarrow$ C

$RESET_i$ = setup.in$_i$! cit.reset $\rightarrow$ setup.out$_i$! nil $\rightarrow$ RESET$_{i+1}$

N.B. This section describes the command set and functionality of the IMS C004 that should be available from revision B silicon onwards. At the time of print, revision A only is available. The *ct.disconnect.output* and *ct.disconnect.link* commands are not implemented and neither is the bit 7 coding after a *ct.enquire* command which will give the output state. Section 4.5 (and section 4.8) gives an example of setting up the IMS C004 dynamically, and assumes that these functions have been implemented.

4.8   CSP description of a 32 stage bidirectional exchange

Section 4.5 described a 32-way bidirectional exchange using occam. This section describes the same system more formally using CSP [CAR Hoare: Communicating Sequential Processes] (figure 4.17).

N.B. Protocol tokens are prefixed by *ct.* for IMS.C004 tokens and *et.* for exchange tokens.

All messages received from *rx[i]* should be preceded by the destination output (*dest*). On receipt of such a message the INPUT.OUTPUT process will request to the CONTROLLER, a bidirectional link path to process *dest*. The CONTROLLER will determine which processes are currently connected to each end of the proposed link. When it is sure that both ends are free, it will set up IMS.C004 and will inform both ends of the new link that a switch has occurred. Note that in this network two channels are placed on each IMS C004 link, one for each direction.

EXCHANGE = IMS.C004 $\|$ CONTROLLER $\|$ ($\|_{j=0..31}$INPUT.OUTPUT$_j$)

The exchange modelled will have an IMS.C004, a CONTROLLER and 32 INPUT.OUTPUT processes.

Controller

CONTROLLER firstly receives a token on *up[0]*. If this is an acknowledge token the rest of the packet is simply consumed. Otherwise depending on whether the token is a request or release token, it proceeds to one of two other processes.

CONTROLLER =   up[0]? token
                          $\rightarrow$ *case*   token = et.ack $\Rightarrow$ (up[0]? any, any $\rightarrow$ CONTROLLER)
                                         token = et.req $\Rightarrow$ DEAL.WITH.REQ
                                         token = et.rel $\Rightarrow$ SETUP.LINK.OR.SEND.NEW.RELEASE
                              esac

DEAL.WITH.REQ firstly receives the source and destination addresses of the requested connection. It then finds out which inputs are already connected to *source* and *dest* and passes information to the relevant INPUT.OUTPUT processes (via a release packet on the daisy chain) to inform them that their outputs are being acquired. Note that if either or both of the addresses are currently free (*state* is *false*) the release packet is stuffed with address *nil*.

DEAL.WITH.REQ =   up[0]? source, dest
                  → c.in! ct.enquire, source → c.out? source.state, current.source.conn
                  → c.in! ct.enquire, dest → c.out? dest.state, current.dest.conn
                  → up[32]! et.rel, source, dest

                  → *case*   (source.state = true) AND (dest.state = true)
                            ⇒ (up[32]! current.source.conn, current.dest.conn → CONTROLLER)
                            (source.state = true) AND (dest.state = false)
                            ⇒ (up[32]! current.source.conn, nil → CONTROLLER)
                            (source.state = false) AND (dest.state = true)
                            ⇒ (up[32]! nil, current.source.conn → CONTROLLER)
                            (source.state = false) AND (dest.state = false)
                            ⇒ (up[32]! nil, nil → CONTROLLER)
                  esac

SETUP.LINK.OR.SEND.NEW.RELEASE firstly receives the rest of the release packet which has visited every INPUT.OUTPUT process. It then examines IMS.C004 (*ct.enquire*) to determine if any changes have been made to the source and dest setup since the release message was sent. If not then the link is set up and an acknowledge packet is transmitted. Otherwise a new release packet is sent.

SETUP.LINK.OR.SEND.NEW.RELEASE =

            up[0]? last.source.conn, last.dest.conn, source, dest
            → c.in! ct.enquire, source → c.out? source.state, current.source.conn
            → c.in! ct.enquire, dest → c.out? dest.state, current.dest.conn
            → (c.in! ct.disconnect.link, current.source.conn, source
                → c.in! ct.disconnect.link, current.dest.conn, dest
                → c.in! ct.link, source, dest → up[32]! et.ack, source, dest → CONTROLLER)
            ⋪ ((last.source.conn = current.source.conn) OR (source.state = false))
                                    AND
              ((last.dest.conn = current.dest.conn) OR (dest.state = false)) ⋫
            (up[32]! et.rel, current.source.conn, current.dest.conn, source, dest → CONTROLLER)

Input.Output

The process INPUT.OUTPUT$_i$ has three states. In all states a message received on *cross.out[i]* from IMS.C004 will be passed directly to its output channel *tx[i]*.

INPUT.OUTPUT$_i$ = INACTIVE$_i$

Initially each process is set up to *inactive*. While in this state the process may receive messages on all three channels. If a message is received preceded by a destination address on *rx[i]*,a request is sent to the CONTROLLER via *up[i]* and the process state is now *pending*. If a token is received from the CONTROLLER via *up[i+1]*, then depending on whether it is a request, release or acknowledge token one of two processes is selected.

> INACTIVE.PASS.RELEASE.PROTOCOL receives the rest of the release packet and since the process in this state is not talking to an output, there is no change of state and the data packet is passed along the daisy chain (*up[i]*).

> INACTIVE.REQ.OR.ACK receives the source and destination addresses and passes the complete packet to the next input.output process. If the packet is a request then there is no change in state, but if the packet is acknowledge and it has a local address (*dest = i*) then the process becomes active and can now talk to *source*.

INACTIVE$_i$ = rx[i]? dest, mess → up[i]! et.req, i, dest → PENDING$_{i,mess}$

> | cross.out[i]? source, mess → tx[i]! source, mess → INACTIVE$_i$

> | up[i+1]? token
> → INACTIVE.PASS.RELEASE.PROTOCOL$_i$
>   ≮ token = et.rel ≯
>   INACTIVE.REQ.OR.ACK$_{i,token}$

INACTIVE.PASS.RELEASE.PROTOCOL$_i$ = up[i+1]? source, dest, addr1, addr2
> → up[i]! et.rel, source, dest, addr1, addr2 → INACTIVE$_i$

INACTIVE.REQ.OR.ACK$_{i,token}$ = up[i+1]? source, dest → up[i]! token, source, dest
> → INACTIVE$_i$
>   ≮ token = et.req ≯
>   (ACTIVE$_{i,source}$ ≮ dest = i ≯ INACTIVE$_i$)

When the process is *pending* it cannot receive any more messages (rx[i]) since it is waiting to send one already. If a token is received from the CONTROLLER via *up[i+1]*, then depending on whether it is a request, release or acknowledge token one of two processes is selected.

> PENDING.PASS.RELEASE.PROTOCOL receives the rest of the release packet and since the process in this state is not talking to an output, there is no change of state and the data packet is passed along the daisy chain (*up[i]*).

> PENDING.REQ.OR.ACK receives the source and destination addresses and passes the complete packet to the next input.output process. If the packet is a request then there is no change in state, but if the packet is acknowledge and it has a local address (*source = i*) then the message is sent (preceded by the source address) and the process goes into *active* state.

PENDING$_{i,m}$ = cross.out[i]? source, mess → tx[i]! source, mess → PENDING$_{i,m}$

> | up[i+1]? token
> → PENDING.PASS.RELEASE.PROTOCOL$_{i,m}$
>   ≮ token = et.rel ≯
>   PENDING.REQ.OR.ACK$_{i,m,token}$

$$\text{PENDING.PASS.RELEASE.PROTOCOL}_{i,m} = \quad up[i+1]?\ source,\ dest,\ addr1,\ addr2 \rightarrow$$
$$up[i]!\ et.rel,\ source,\ dest,\ addr1,\ addr2 \rightarrow \text{PENDING}_{i,\,m}$$

$$\text{PENDING.REQ.OR.ACK}_{i,m,token} = \quad up[i+1]?\ source,\ dest \rightarrow up[i]!\ token,\ source,\ dest$$
$$\rightarrow \text{PENDING}_{i,m}$$
$$\not< token = et.req \not>$$
$$((cross.in[i]!\ i,m \rightarrow \text{ACTIVE}_{i,\,dest}) \not< source = i \not> \text{PENDING}_{i,m})$$

When the process is *active*, a message received (*rx[i]*) will either be passed straight to IMS.C004 (*cross.in[i]*) (if destination address is unchanged) replacing *dest* with *i*, or the process will request a new output to talk to and switch to *pending* state. If a token is received from the CONTROLLER via *up[i+1]*, then depending on whether it is a request, release or acknowledge token one of two processes is selected.

ACTIVE.PASS.RELEASE.PROTOCOL receives the rest of the release packet and passes it along the daisy chain. If the packet addresses indicate that this connection should be released it becomes *inactive*. Otherwise the state is preserved.

ACTIVE.REQ.OR.ACK receives the source and destination addresses and passes the complete packet to the next input.output process. If the packet is a request then there is no change in state, but if the packet is acknowledge and it has a local address (*dest = i*) then its connection address is changed to *source*.

$$\text{ACTIVE}_{i,d} = \quad rx[i]?\ dest,\ mess \rightarrow$$
$$(cross.in[i]!\ i,\ mess \rightarrow \text{ACTIVE}_{i,d}) \not< d = dest \not> (up[i]!\ et.req,\ i,\ dest \rightarrow \text{PENDING}_{i,\,mess})$$

$$|\ cross.out[i]?\ source,\ mess \rightarrow tx[i]!\ source,\ mess \rightarrow \text{ACTIVE}_{i,d}$$

$$|\ up[i+1]?\ token$$
$$\rightarrow \text{ACTIVE.PASS.RELEASE.PROTOCOL}_{i,d}$$
$$\not< token = et.rel \not>$$
$$\text{ACTIVE.REQ.OR.ACK}_{i,d,token}$$

$$\text{ACTIVE.PASS.RELEASE.PROTOCOL}_{i,d} =$$
$$up[i+1]?\ source,\ dest,\ addr1,\ addr2 \rightarrow$$
$$up[i]!\ et.rel,\ source,\ dest,\ addr1,\ addr2 \rightarrow$$
$$\rightarrow \text{INACTIVE}_i$$
$$\not< ((dest = i)\ \text{AND}\ (source \neq d))\ \text{OR}\ (addr1 = i)\ \text{OR}\ (addr2 = i) \not>$$
$$\text{ACTIVE}_{i,d}$$

$$\text{ACTIVE.REQ.OR.ACK}_{i,d,token} = \quad up[i+1]?\ source,\ dest \rightarrow up[i]!\ token,\ source,\ dest$$
$$\rightarrow \text{ACTIVE}_{i,d}$$
$$\not< token = et.req \not>$$
$$(\text{ACTIVE}_{i,\,source} \not< dest = i \not> \text{ACTIVE}_{i,d})$$

5 Module motherboard architecture

5.1 Introduction

INMOS transputer modules are designed to form the building blocks of parallel processing systems. They consist of printed circuit boards in a range of sizes which typically hold a member of the transputer family of processors, some memory and perhaps some application specific circuitry. A module needs only a 5 volt power supply and a 5MHz clock to operate. These are supplied to the module through pins on the periphery of the board. Other pins bring out the transputer's serial links and reset, analyse and error signals. Some modules can control a subsystem of other modules through another set of pins. The *Dual-In-Line Transputer Modules (TRAMs)* document provides a complete specification of INMOS transputer modules.

In order to use modules as parallel processing building blocks INMOS has developed a range of motherboards. While these boards provide access to transputers from a number of different host machines, they have a common architecture to allow control and interconnection of potentially large numbers of transputers. This document describes the generic architecture of module motherboards. It is recommended that this specification is followed when designing in order to preserve compatibility with INMOS module motherboards.

5.2 Module motherboard architecture

The INMOS range of module motherboards has a common architecture making it easy to build and configure systems consisting of large numbers of transputer modules. The goals aimed at in the design of the module motherboards, and the architecture developed to achieve them, are described below.

5.2.1 Design goals

The main goals aimed at in the design of module motherboards are:

- To be able to build systems with any number of transputer modules in any combination of type or size

- To be able to build a variety of different kinds of network (e.g. arrays, trees, cubes, etc.)

- Enable any number of motherboards to be chained together

- Make transputer link connections easily configurable by software

- To be able to run test and applications programs on transputers without first configuring links

- Provide a standard hardware interface to configuration and applications software

- Allow hierarchical control of systems of transputers

- Make the transputer hardware and software independent of the host system

5.2.2 Architecture

In order to achieve the design goals outlined above, a standard architecture is adopted for all module motherboards. The rest of this document describes the motherboard architecture in detail, but the salient features are given below.

- The modules in a network are connected in a pipeline using two links from each module

- The remaining links from each module are taken to IMS C004 programmable link switches

- A number of links are taken from IMS C004s to edge connectors for wiring to other boards

- Each IMS C004 is controlled by an IMS T212 transputer

- The IMS T212s are connected in a separate pipeline

- The first module in the pipeline on a particular motherboard can control a subsystem of other transputers that may reside on the same motherboard, another motherboard or may be distributed across a number of boards

- An interface may be provided to enable a non-transputer based host system to control and communicate with a motherboard

5.3   Link configuration

Transputers communicate with each other via serial links operating at 10 or 20Mbits/s. The module motherboard architecture facilitates the interconnection of links between transputer modules by providing a standard hardware link configuration and allowing software configuration using IMS C004 programmable link switches. Links should be interconnected by properly terminated transmission lines (PCB trace or cable) having a characteristic impedance of 100Ω. INMOS Technical note 18, *Connecting INMOS links*, gives full details on all aspects of connecting links.

5.3.1   Pipeline

Each module resides in a *module slot* which provides two sockets that take the 16 pins of a size 1 module. A motherboard may have any number of module slots, determined only by the physical size of the board. The slots are numbered starting at slot 0.

All the modules on a motherboard are connected in a pipeline as shown in figure 5.1. Link 2 of the module in
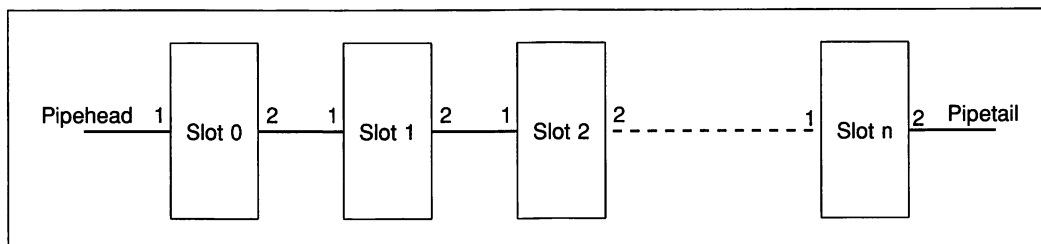
Figure 5.1 Module pipeline

slot 0 is connected to link 1 of slot 1 and so on for the rest of the pipeline. Link 1 of module slot 0 *(Pipehead)* and link 2 of the last module slot *(Pipetail)* are brought out to an edge connector thus enabling the pipelines of any number of boards to be chained together by connecting Pipehead of one board to Pipetail of the next. See figure 5.2.
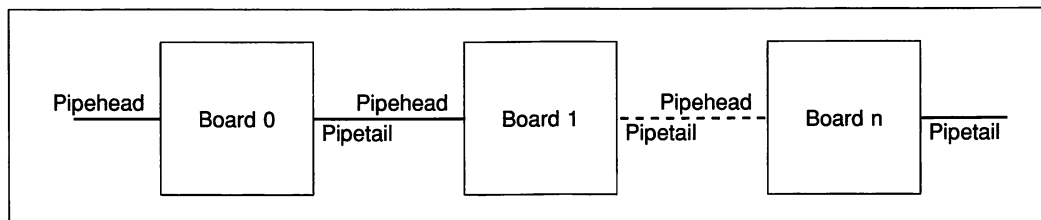
Figure 5.2 Module pipeline on several boards

Some applications may not require a full complement of modules or may use size 2 or larger modules which take up more than one slot, but use only one slot for electrical connection. In either case the pipeline will be

broken unless steps are taken to keep it intact. A *pipe jumper* is a small connector used for this purpose. See figure 5.3. It plugs into an unused module slot and connects link 1 of that slot to link 2 of the same slot, thus preserving the pipeline.
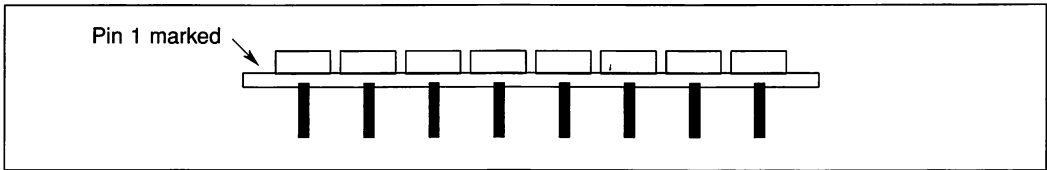
Figure 5.3 Pipe jumper

5.3.2   IMS C004 link configuration

An IMS C004 programmable link switch is used for software configuration of links. This device is a crossbar switch which can handle up to 32 links. It can connect any of the 32 link inputs to any of the 32 link outputs under software control from a separate configuration link.

Links 0 and 3 of each module are taken to an IMS C004 or a number of IMS C004s, depending on the number of links. Links may be taken from an IMS C004 to an edge connector to allow links from one motherboard to be connected to those of another.

The number of IMS C004s required on a particular motherboard depends on the number of modules the board can hold. The exact arrangement of IMS C004 links is not specified here in order to give the designer maximum flexibility for his particular application. **The only restriction is that links 0 and 3 of each module are taken to a C004.** This may be done in a number of ways. For example:

- Link 0s may be taken to one IMS C004 or a set of IMS C004s; link 3s may be taken to another IMS C004 or a set of them

- Both Link 0s and link 3s may be taken to the same IMS C004(s)

- LinkOut0s and LinkOut3s may be connected to an IMS C004 or a set of the same, while LinkIn0s and LinkIn3s are taken to another IMS C004 or a set of them

5.3.3   T212 pipeline and C004 control

Each IMS C004 on a motherboard is controlled from an IMS T212 16-bit transputer as shown in figure 5.4. An IMS T212 can control up to two IMS C004s via its links 0 and 3. Links 1 and 2 of each IMS T212 are used to connect the transputers in a *configuration pipeline*. Link 1 of the first IMS T212 on the board is taken to an edge connector designated *ConfigUp*; link 2 of the last IMS T212 in the board's configuration pipeline is also taken to an edge connector designated *ConfigDown*. In this way the configuration pipelines of any number of motherboards may be chained together by connecting ConfigDown of one board to ConfigUp of the next, enabling a network of transputer modules spread over several boards to be configured from software.

The IMS C004 configuration data may come from software running on a module residing on the first motherboard in the system. It is therefore necessary to be able to connect a link of that module to the board's configuration pipeline. A jumper provides the option of connecting link 1 of the first IMS T212 in the configuration pipeline *either* to ConfigUp *or* to link 1 of module slot 0. In the latter, the jumper also disconnects PipeHead on the edge connector from slot 0 link 1. This is shown diagrammatically in figure 5.5.

5.3.4   Software link configuration

The hardware configuration described in Sections 5.3.2 and 5.3.3 provides the standard architecture recognised by the *Module Motherboard Software (MMS)*, a software package available from INMOS which allows easy configuration of the IMS C004 link connections.

The MMS takes a list of link connections that are hardwired on the board together with a list of the required
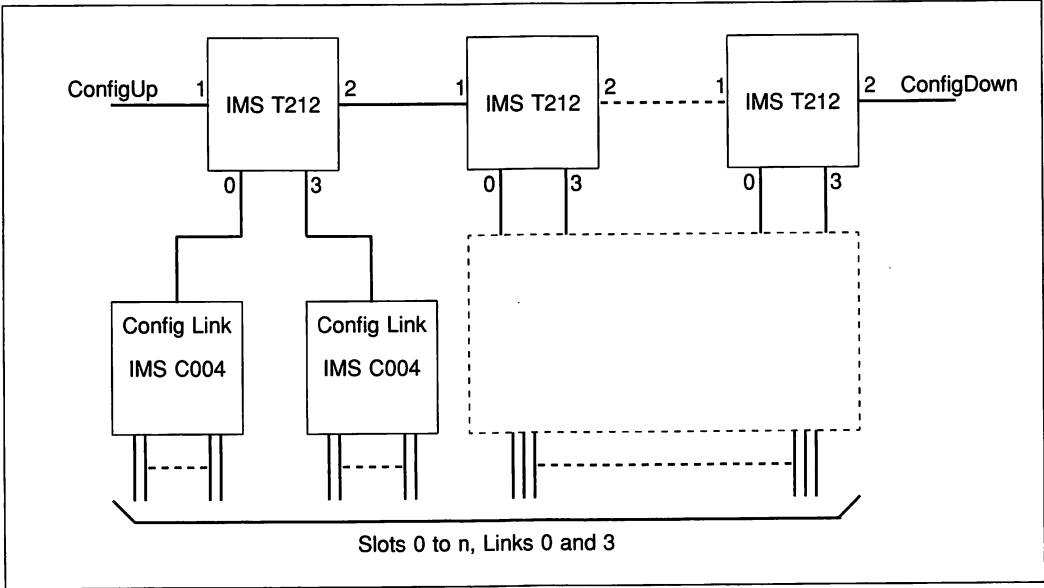
Figure 5.4 IMS C004 control by a pipeline of IMS T212s
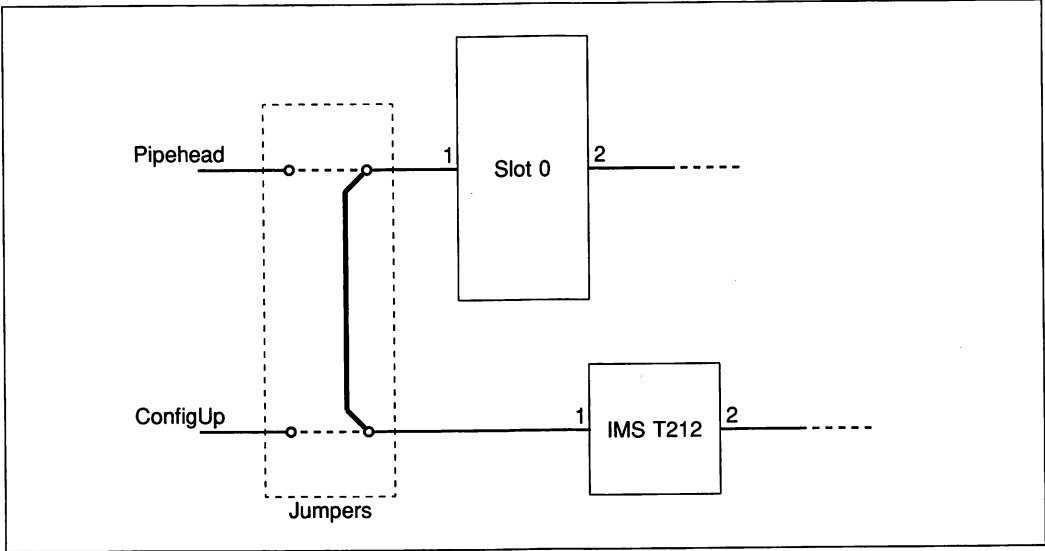
Figure 5.5 ConfigUp/Pipehead jumper

'softwired' connections and generates the configuration details for each IMS C004.

For each board in the system, the user can:

- Connect link 0 of any module to link 3 of any module
- Connect link 0 or link 3 of any module to an edge connector link

- Connect an edge connector link to another edge connector link

The MMS is described in detail in the *MMS2 User Guide*.

5.4   System control

The subsystem control function of the module motherboard architecture allows hierarchical control of networks of transputers. It enables a module capable of driving a subsystem to reset or analyse a network of modules and to handle errors in the network. The driving module can itself form part of a network which is controlled by another module. In this way a hierarchy of control is made possible.

Each module on a motherboard requires a 5MHz clock. The module motherboard specification provides a scheme for distributing the clock signal from a single crystal oscillator to all the modules on a motherboard.

5.4.1   Reset, analyse and error

Three signals are provided by transputers for the purpose of allowing system control: *Reset*, *Analyse* and *Error*. The **Reset** and **Analyse** inputs enable the transputer to be initialised or halted in a way which preserves its state for subsequent analysis. The transputer **Error** signal is connected directly to the processor's Error flag. See the *Transputer Reference Manual* for a detailed description of these signals.

A transputer module has a similar set of signals: module **Reset** and **Analyse** are connected directly to the respective pins on the transputer; the transputer **Error** pin is taken to a transistor on the module to produce an open collector **notError** signal that can be wire-ORed with the **notError** signals of other modules.

Some modules are capable of controlling a subsystem of other modules. They have three extra pins: **SubSystemReset**, **SubSystemAnalyse** and **notSubSystemError**, which are controlled by the on-module transputer through latches in memory. These pins are connected to the **Reset**, **Analyse** and **notError** pins of the modules in the subsystem being controlled. The subsystem can then be reset or analysed by asserting the relevant signal of the subsystem controller module. The subsystem's ORed **notError** signal can also be monitored by the controlling module.

5.4.2   Up, down and subsystem

A module motherboard has three ports that provide hierarchical control: Up, Down and subsystem (see figure 5.6). Each port appears at an edge connector and has three active-low signals: **notReset**, **notAnalyse** and **notError**. A board is able to control a subsystem of other boards by connecting its subsystem port to the Up port of the next board. Boards in a subsystem are chained together by connecting the Down port of one board to the Up port of the next board. A board within a subsystem is in turn able to control another network through its subsystem port.

Figure 5.7 shows how a board can be connected to a subsystem of boards.

The **notReset** and **notAnalyse** signals flow from subsystem of one board to Up of the next board. From there, they go directly to Down. They are also logical ORed with that board's subsystem reset and analyse latches and then pass to the subsystem port. The **notError** signal passes from a board through its Up port. If it is connected to the Down port of the board above, it is logical ORed with that board's Error signal and passed to the Up port. If it goes to the subsystem port of the board above, the Error signal is not passed on, but is handled by that board. (Figures 5.10, 5.11 and 5.12 show the module motherboard system control logic.)
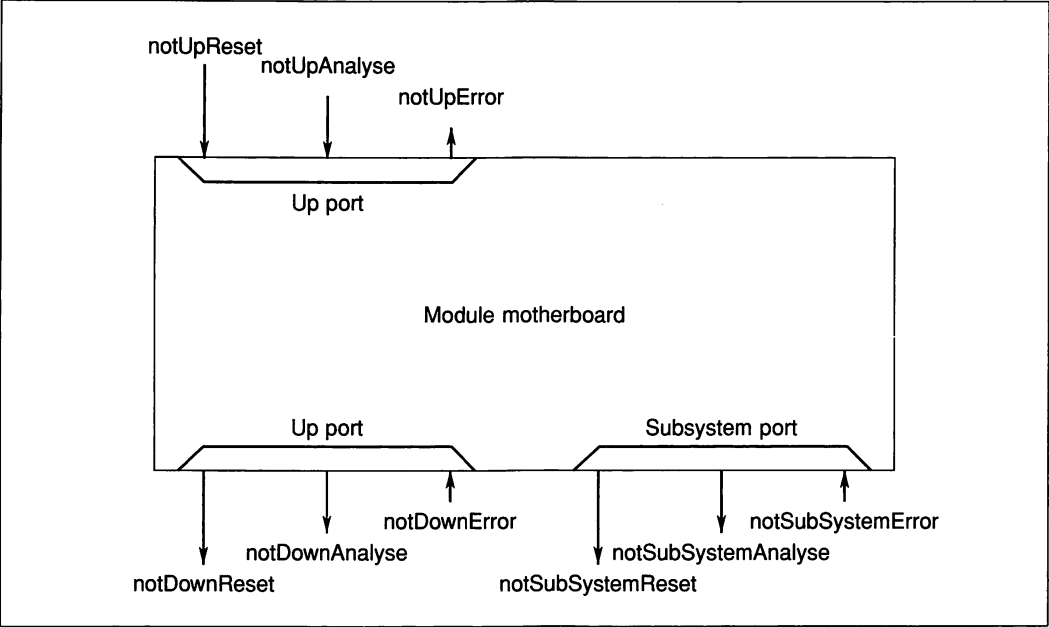
Figure 5.6 Up, down and subsystem
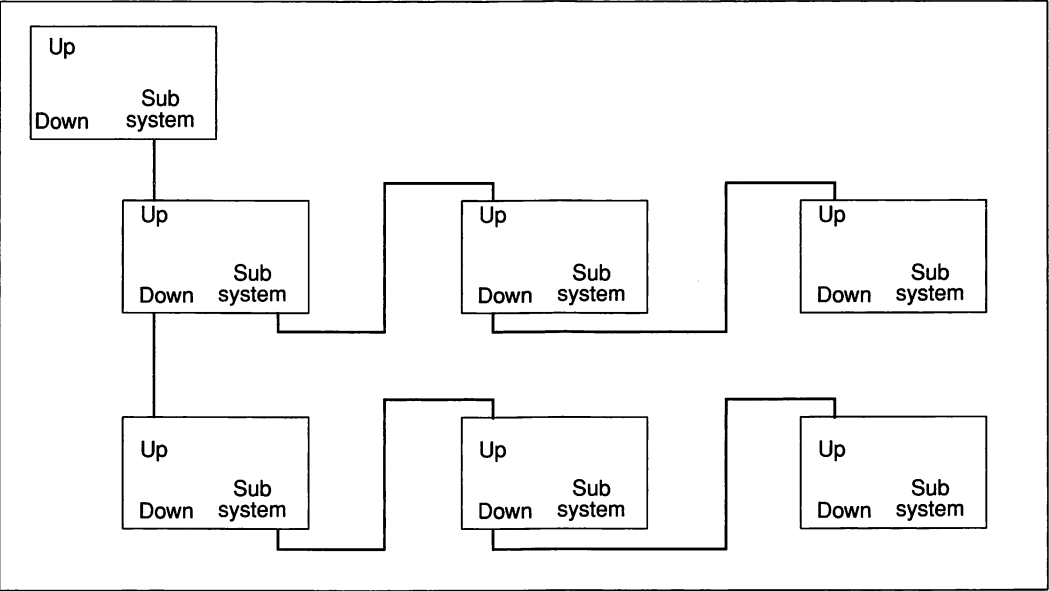
Figure 5.7 Controlling a subsystem of boards

5.4.3   Source of control

If there are *n* slots on a motherboard, modules in slots 1 to *n* may be controlled from either the Up port (or a host machine if the motherboard has an interface to one, see Section 5.5) or may be part of a subsystem controlled by a suitable module in slot 0. The source of control is determined by a jumper or switch, as shown in figure 5.8.
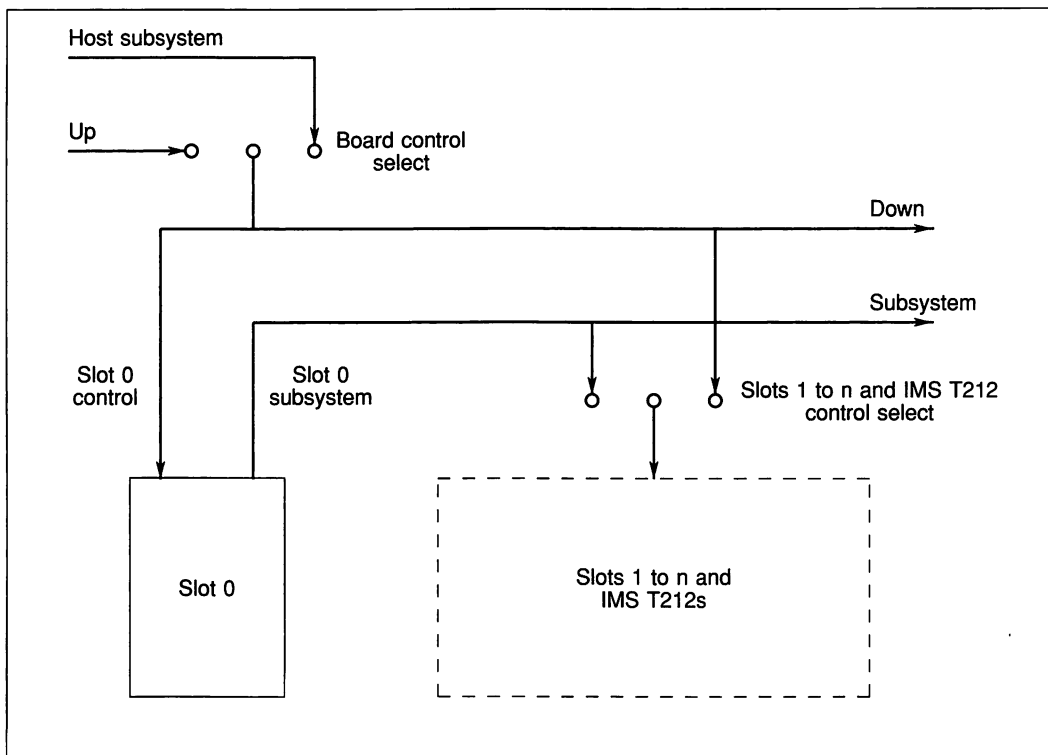
Figure 5.8 Source of control

The on-board IMS T212(s) may be reset and analysed from the same source that controls slots 1 to *n*. The **Error** pin of the IMS T212(s) is not connected.

A power-on reset circuit is required for the IMS C004(s) on board. An IMS C004 may then be reset at power-on or by the IMS T212 controlling it. Each IMS T212 has a latch mapped into its memory space. See figure 5.9. This enables software running on the IMS T212 to reset the IMS C004 either by setting the latch or by sending a reset message to the IMS C004 Configuration link.
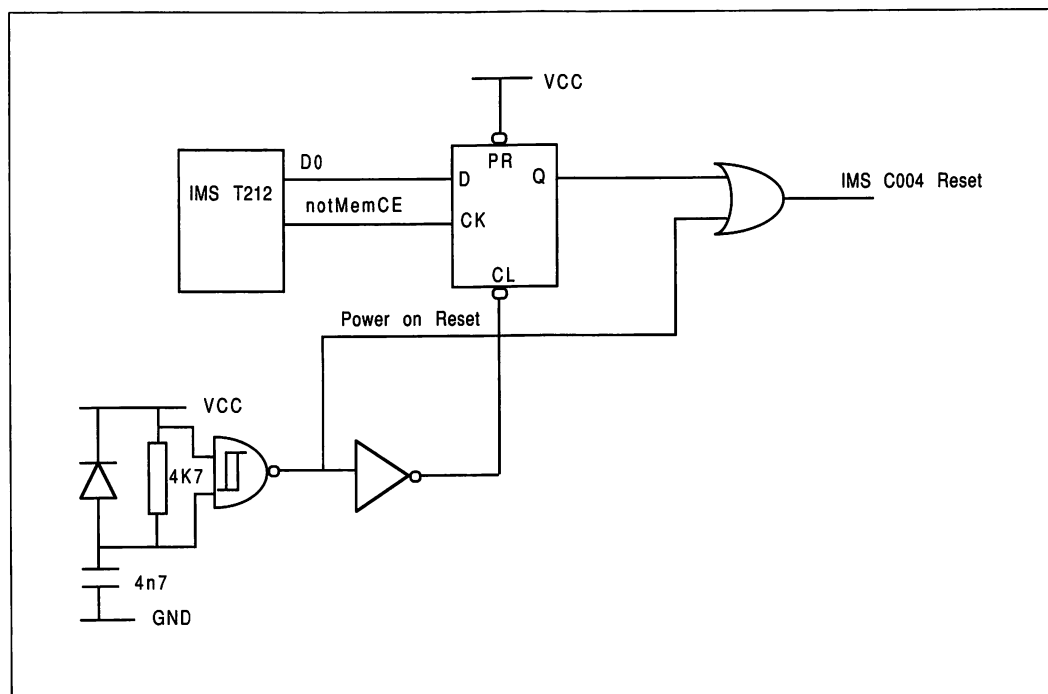
Figure 5.9 IMS C004 reset circuit

Figures 5.10, 5.11 and 5.12 show the logic required for Reset, IMS C004 Reset, Analyse and Error, respectively. These diagrams provide a logical description only: the actual implementation is left to the individual designer. It is important, however, to include the passive components indicated in the diagrams. The 1K pull-up resistors on the **notUpReset**, **notUpAnalyse**, **notDownError** and **notSubSystemError** signals are necessary to ensure that if these signals are unconnected they are not left floating, but are deasserted. The 4K7 pull-up resistors are required to wire-OR the open collector **notError** signals from the module slots. Note that the *Dual-In-Line Transputer Modules (TRAMs)* document specifies a maximum of ten **notError** signals should be wire-ORed together. The combination of each 100Ω resistor and 100nF capacitor filters out noise on the **notUpReset**, **notUpAnalyse**, **notDownError** and **notSubSystemError** signals coming from off the board.

To improve noise rejection, it is recommended that Schmitt gates are used to receive signals from other boards. These gates should use bipolar technology (e.g low power Schottky 74LS series TTL). It is also recommended that gates driving signals off the board are capable of providing a full output voltage swing from 0V to 5V, e.g. HCT series gates.

The Reset logic (figure 5.10) uses the **Board Control Select** switch and multiplexer to select whether Slot 0 and the Down port are reset from the Up port or from the host. The **Slots 1 to n & IMS T212 Control Select** switch and multiplexer determine whether Slots 1 to n and the IMS T212s are reset from the Slot 0 subsystem port or from the Up port or the host. A similar arrangement is used for the Analyse logic (figure 5.11).

In the Error logic (figure 5.12), the **Slots 1 to n & IMS T212 Control Select** switches and multiplexers select whether **notError** from Slots 1 to n is passed either to the Slot 0 subsystem port or to the Up port or the host. The **Board Control Select** switch and decoder determine whether **Slots 1 to n notError, notDownError** or **notSlot0Error** are passed to the Up port or to the host.

Board Control Select and **Slots 1 to n & IMS T212 Control Select** correspond to the conceptual switches in figure 5.8.
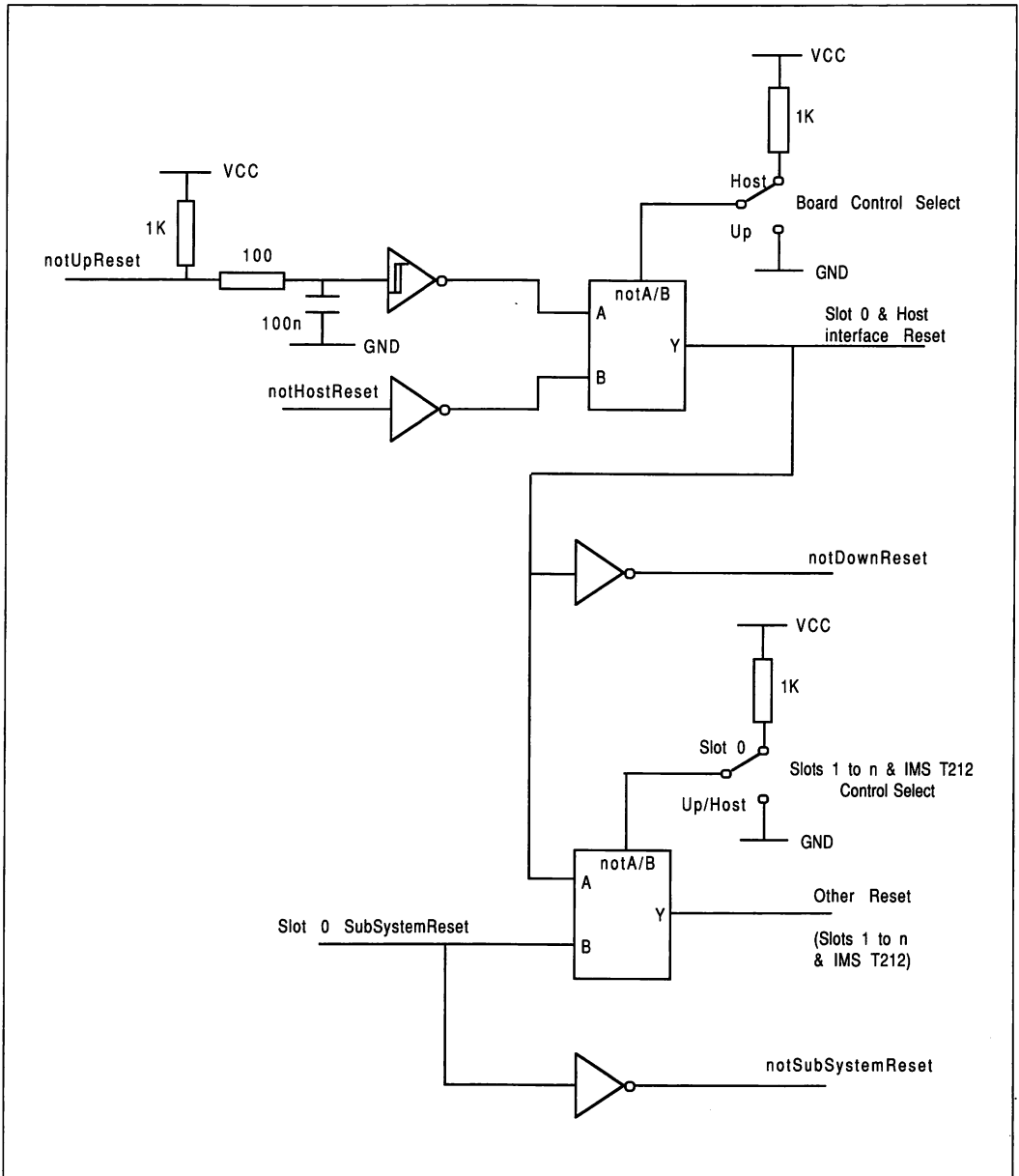
Figure 5.10 Reset logic
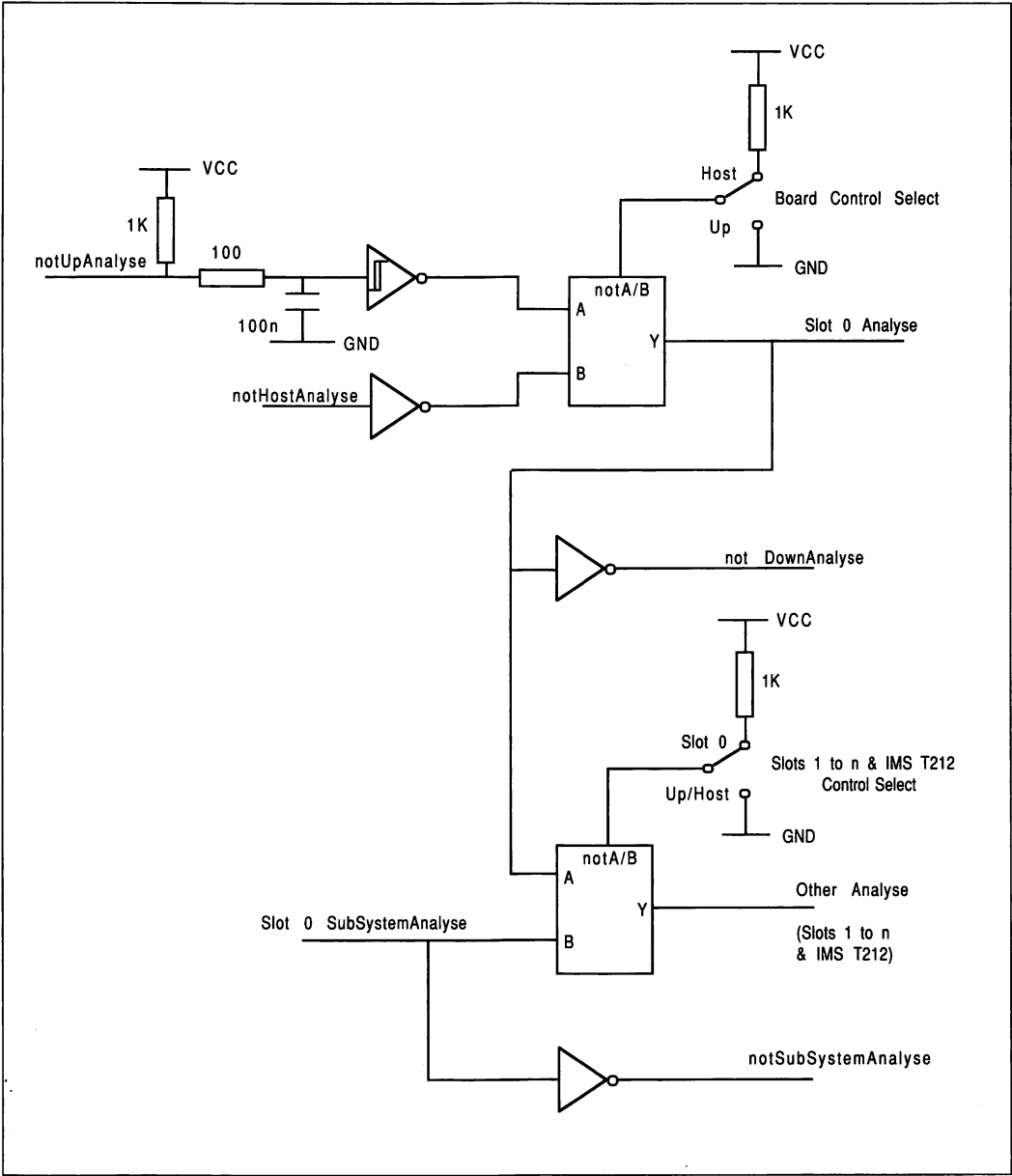
Figure 5.11 Analyse logic
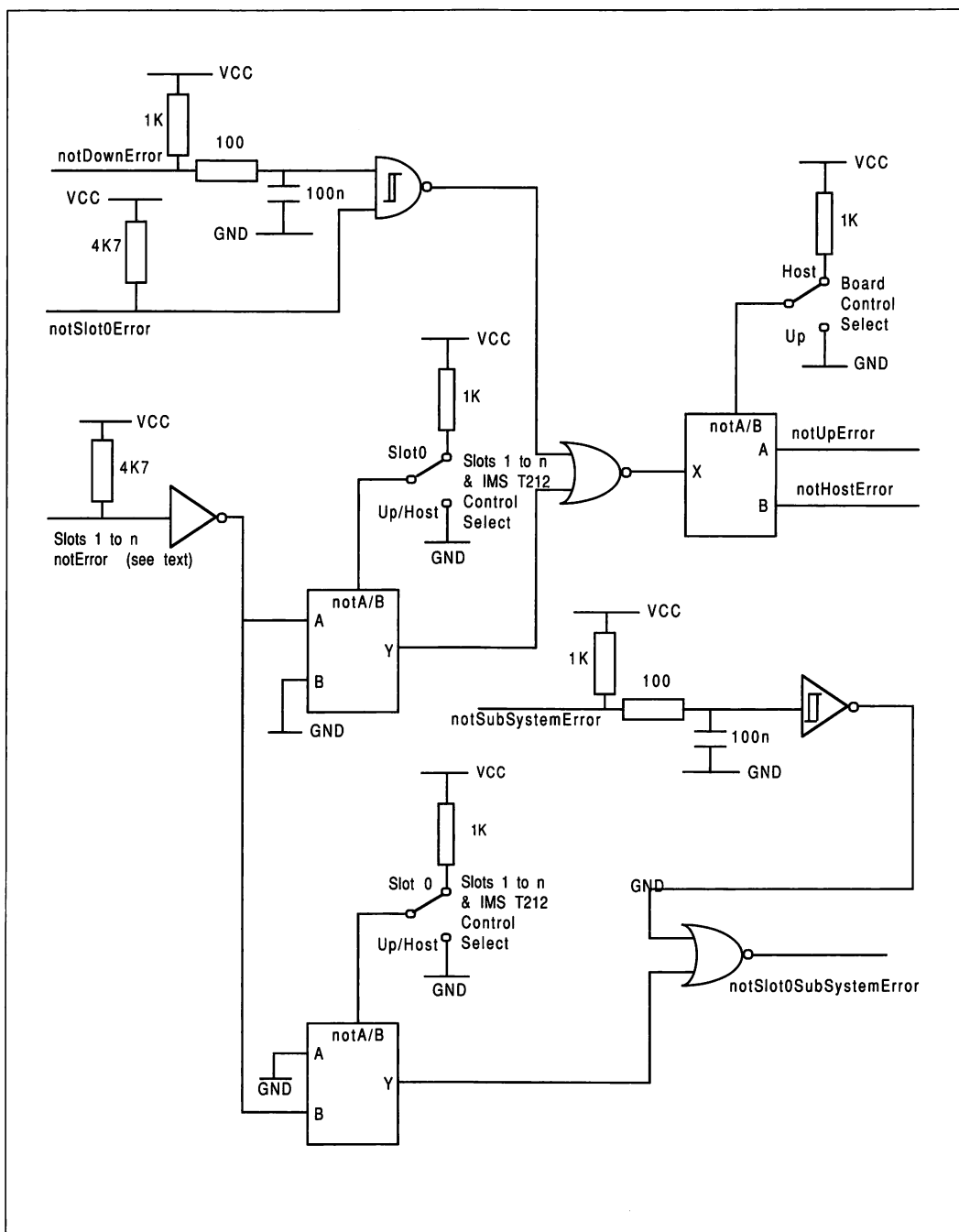
Figure 5.12 Error logic

5.4.4   Clock

A 5MHz, TTL compatible clock signal is required for each module slot, IMS T212 and IMS C004 on board. Since the clock must be distributed to a number of modules and devices the buffering scheme shown in figure 5.13 is used to minimise distortion of the clock waveform caused by excessive loading and transmission line effects. This is a *star* configuration and it may be extended indefinitely by adding more buffers at the star points which may drive further buffers, and so on until the required number of clock signals are derived. The length of any pcb trace carrying a clock signal should be limited to 30cm.
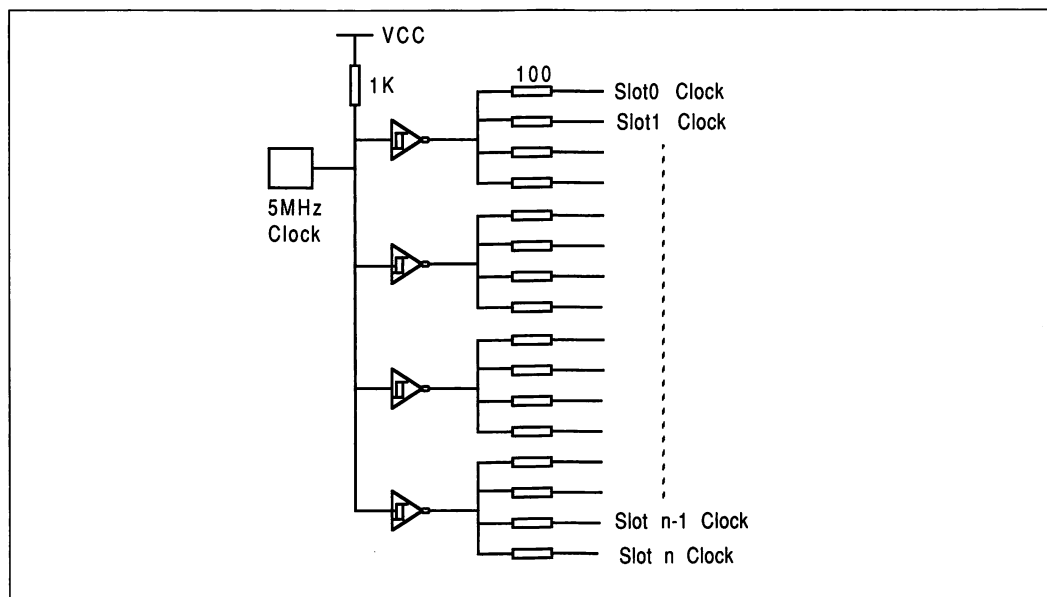
Figure 5.13 Clock distribution

5.5   Interface to a separate host

Some module motherboards may require an interface to a host machine or system that is not transputer based, e.g. the IBM PC, VMEbus or Futurebus. Because the implementation of the interface is specific to the host system, it is not defined here. However, it should allow the system to access the module pipeline and control a subsystem of modules.

5.5.1   Link interface

The host system accesses the module pipeline via Slot 0 Link 0, as shown in figure 5.14. It is beyond the scope of this document to define the implementation of the host to link interface, but it might consist of an INMOS link adapter, the registers of which may be mapped into the host's address space, or it may involve the use of dual-ported RAM shared between the host and a transputer.

The interface must be capable of interrupting the host when a data transfer in either direction has been completed.
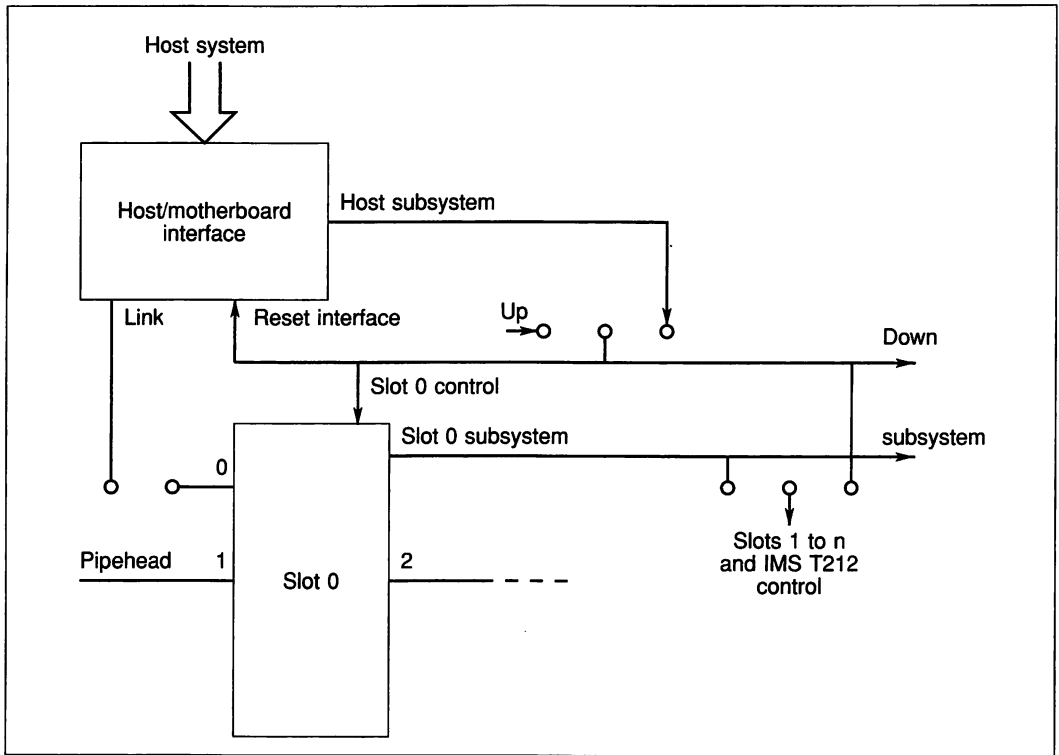
Figure 5.14 Host to motherboard interface

5.5.2   System control interface

The host system must be able to control a network of modules. This is made possible by the provision of latches mapped into the host's memory. There are three latches: Reset, Analyse and Error, which correspond to the **notHostReset, notHostAnalyse** and **notHostError** signals of the *HostSubSystem* port shown in figure 5.14. The Reset and Analyse latches are mapped into successive locations of host memory. Reset and Analyse are write only by the host; the Error latch is read only and shares the same address as the Reset latch.

Writing a '1' into bit 0 of the Reset latch asserts **notHostReset**;
Writing a '0' into bit 0 of the Reset latch deasserts **notHostReset**.

Writing a '1' into bit 0 of the Analyse latch asserts **notHostAnalyse**;
Writing a '0' into bit 0 of the Analyse latch deasserts **notHostAnalyse**.

A '1' read in bit 0 of the Error latch indicates that **notHostError** is asserted;
A '0' read in bit 0 of the Error latch indicates that **notHostError** is deasserted.

The host to motherboard link interface is reset by the same source as Slot 0, i.e. the Up port or the HostSubSystem port.

5.5.3   Interrupts

The host to subsystem interface must be capable of generating an interrupt to the host when certain events occur on the motherboard. These include:

- Completion of transfer of data from the host to the motherboard

- Completion of transfer of data from the motherboard to the host

- Error in subsystem indicated by **notHostError** being set

Other system specific conditions may also generate an interrupt, e.g. if DMA is used to transfer data between the host and motherboard, the end of a DMA cycle may trigger an interrupt.

The host may select which conditions cause an interrupt by setting bits in a register or registers on the motherboard, mapped into the address space of the host. Other registers hold status information that can be read by the host to determine the source of an interrupt.

5.6   Mechanical considerations

The size and shape of a module motherboard is determined by its application. However, there are a number of mechanical constraints which must be adhered to in order to maintain compatibility between different modules and motherboards.

The size and spacing of module slots must conform to the mechanical specification in the *Dual-In-Line Transputer Modules (TRAMs)* document, the main points of which are reiterated here.

5.6.1   Dimensions

In the following, dimensions are quoted in inches for PCB length, width and related dimensions; all other dimensions are quoted in millimetres.

Width and length

The basic size of a TRAM is a very wide 16 pin DIP, with 3.3" between the two rows of pins. These TRAMs fit on a 3.6" pitch on their length, and a 1.1" pitch on their width. Extra length is added beyond the pins to hold the pins, to provide for mechanical fixing, and to polarise the module shape. Modules can be made larger than the standard size by keeping the 3.3" between pins and using two or more sets of the 16 pins. They can be made smaller than the standard size, down to a 16 pin DIP with 0.6" between the two rows of pins, or 1.5" between the pins. These sizes will normally be used for single chip modules or hybrids.

The top drawing in figure 5.15 shows a Size1 module and how the jigsaw pattern fits together between adjacent modules. The lower drawing in figure 5.15 shows the various sizes of TRAM. Detailed dimensions of the different sizes are given in the *Dual-In-Line Transputer Modules (TRAMs)* document.
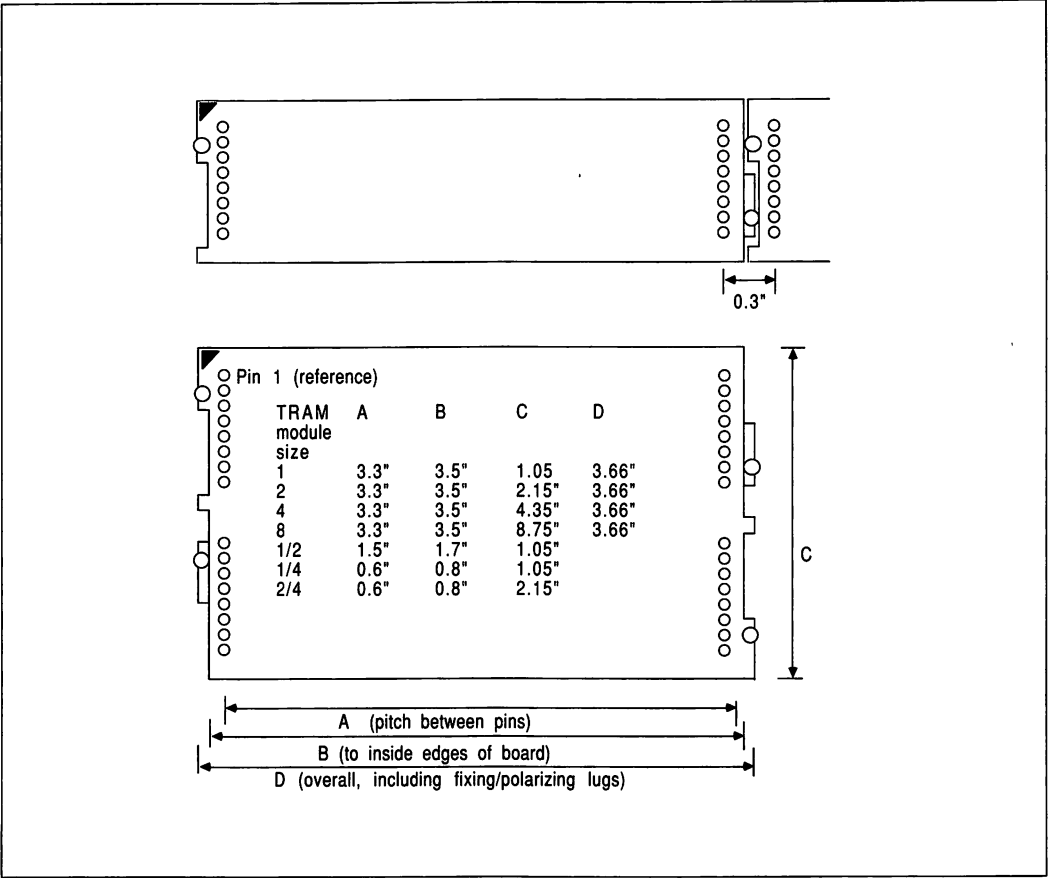
Figure 5.15 Transputer module sizes

Vertical dimensions

The height specifications, both above and below the TRAM PCB, are shown in figure 5.16a. Figure 5.16b shows a module with these dimensions plugged into a motherboard.

Figure 5.16c shows a TRAM above components on a motherboard and the overall component height is 13.7mm, which is within normal specifications for motherboards on 0.8" centres.

It is recommended that any component reaching a maximum specified height has an insulating surface.

To provide the spacing shown in figure 5.16c, the TRAM pins are implemented as a stackable socket, and an extra stackable socket is used between the motherboard socket and module pin.

Figure 5.16d shows an alternative component height which meets the 13.7mm overall height if the module is not above components on a motherboard.

Figure 5.16e shows two modules stacked.

Note that the datum for component heights on both sides of the TRAM is the component side surface. This datum is also used for the stackable socket to minimize tolerance buildup.
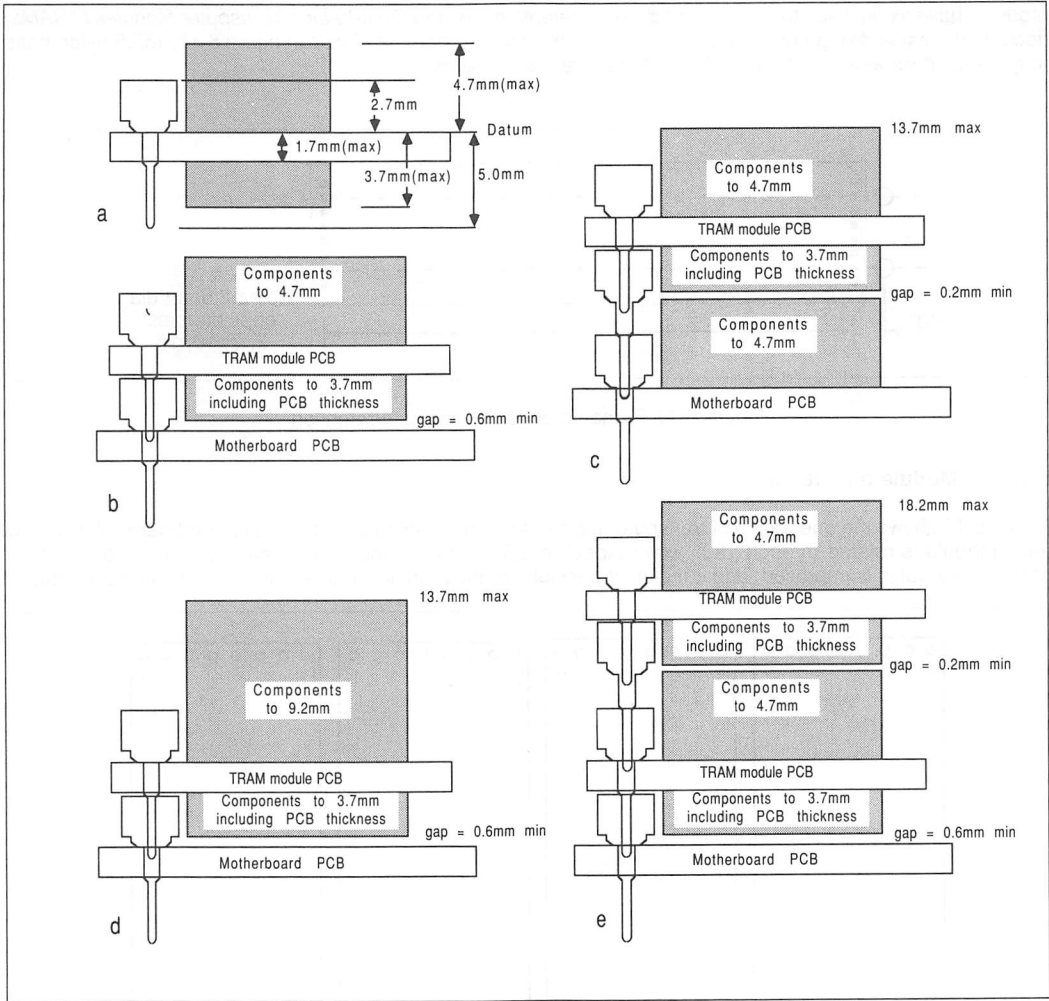
Figure 5.16 Component heights

5.6.2   Motherboard sockets

The TRAM pins/stackable sockets defined in the *Dual-In-Line Transputer Modules (TRAMs)* document will plug into any standard IC socket. To meet the component heights given in figure 5.16, the stackable socket must also be used on the motherboard.

Motherboard sockets for the Slot 0 subsystem signals should be the 0.38mm or 0.4mm sockets referred to in the *Dual-In-Line Transputer Modules (TRAMs)* document.

5.6.3   Mechanical retention of TRAMs

Vibration tests have shown that in a normal office or laboratory environment, the TRAMs remain plugged into their sockets. In transit, however, or in an environment where there is vibration, some form of mechanical retention may be necessary.

Modules have fixing holes to facilitate mechanical retention, see the *Dual-In-Line Transputer Modules (TRAMs)* document. Similar fixing holes should be drilled in the motherboard as shown in figure 5.17. M2.5 nylon bolts may be used between these fixing holes to secure the modules.
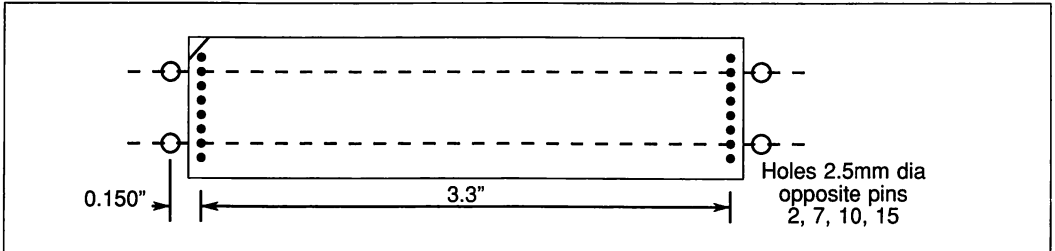
Figure 5.17 Fixing holes for mechanical retention

5.6.4   Module orientation

Figure 5.18 shows the orientation of transputer modules when mounted in slots on a motherboard. Notice how each module is rotated through 180° with respect to adjacent modules. This serves two purposes: cooling of Size 1 modules is improved; and it makes it possible to have Single-In-Line modules at some future date.
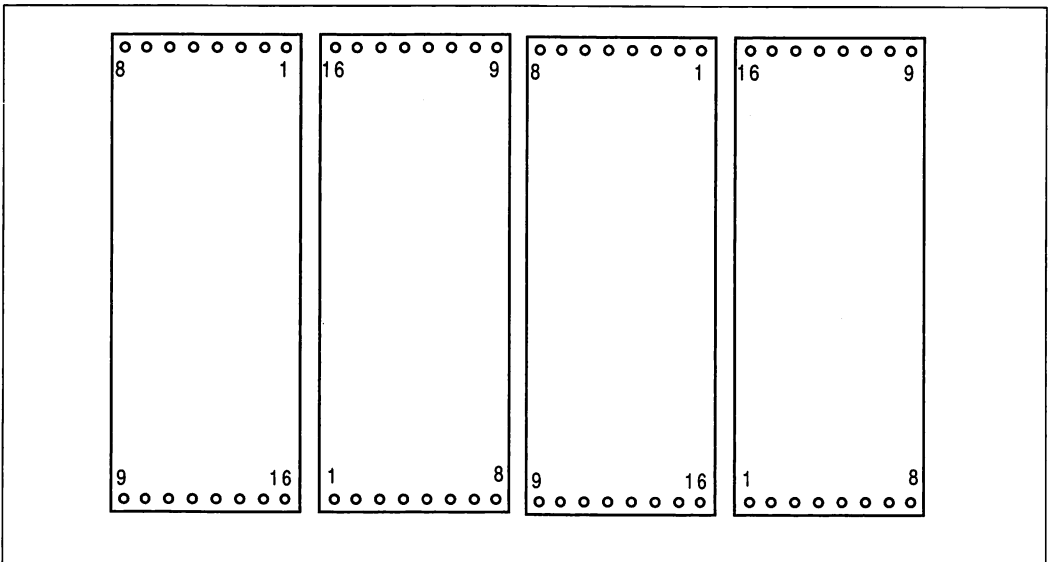
Figure 5.18 Orientation of module slots

5.7   Edge connectors

Connectors are necessary to enable links and system control signals to be taken from a motherboard to other boards. Several types of connector have been used on INMOS module motherboards.

The IMS B008 module motherboard for the IBM PC uses a 37-way D-type connector, the pin-out of which is shown in figure 5.19.

notUpReset	20	1	GND
notUpError	21	2	notUpAnalyse
EdgeLinkIn0	22	3	EdgeLinkOut0
EdgeLinkIn1	23	4	EdgeLinkIn1
EdgeLinkOut2	24	5	GND
EdgeLinkOut3	25	6	EdgeLinkIn2
EdgeLinkOut4	26	7	EdgeLinkIn3
GND	27	8	EdgeLinkIn4
EdgeLinkIn5	28	9	EdgeLinkOut5
EdgeLinkIn6	29	10	EdgeLinkOut6
EdgeLinkIn7	30	11	EdgeLinkOut7
ConfigUpLinkOut	31	12	GND
PipeHeadLinkOut	32	13	ConfigUpLinkIn
notSubSystemReset	33	14	PipeHeadLinkIn
notSubSystemError	34	15	notSubSystemAnalyse
PipeTailLinkIn	35	16	PipeTailLinkOut
ConfigDownLinkIn	36	17	ConfigDownLinkOut
notDownAnalyse	37	18	notDownReset
		19	notDownError

Figure 5.19 37-way D-type connector

This connector provides up to twelve links (including ConfigUp, ConfigDown, PipeHead and PipeTail), plus Up, Down and Subsystem ports. A cable suitable for connecting IMS B008s together is shown diagrammatically in figure 5.20.

The IMS B012 is a module motherboard in double extended Eurocard format. It has two 96-way DIN 41612 connectors. The bottom connector (P2) provides connections for eight links (including ConfigUp, ConfigDown, PipeHead and PipeTail) and Up, Down and SubSystem ports. Table 5.1 shows the general pinout adopted by INMOS for such a connector, making it suitable for use with module motherboards while preserving compatibility with the the rest of the INMOS range of boards. The pins marked *Spare* and *Spare link* may be used for signals and links specific to a particular application. The *IMS B012 User Guide and Reference Manual* describes how these pins are used on the IMS B012.

The top connector (P1) of the IMS B012 is a DIN 41612 connector that takes a special mini-backplane to provide connections to 32 links. See figure 5.21 for the mechanical details and Table 5.2 for the pinout of this connector. On the IMS B012, the P1 connector is used to bring out links from the board's two IMS C004s. See the *IMS B012 User Guide and Reference Manual* for details. The mini-backplane is available from Varelco, part number 07-8258-0940-01-00. Both the P1 and P2 connectors are used with the INMOS Link and Reset cables provided with most INMOS board products.
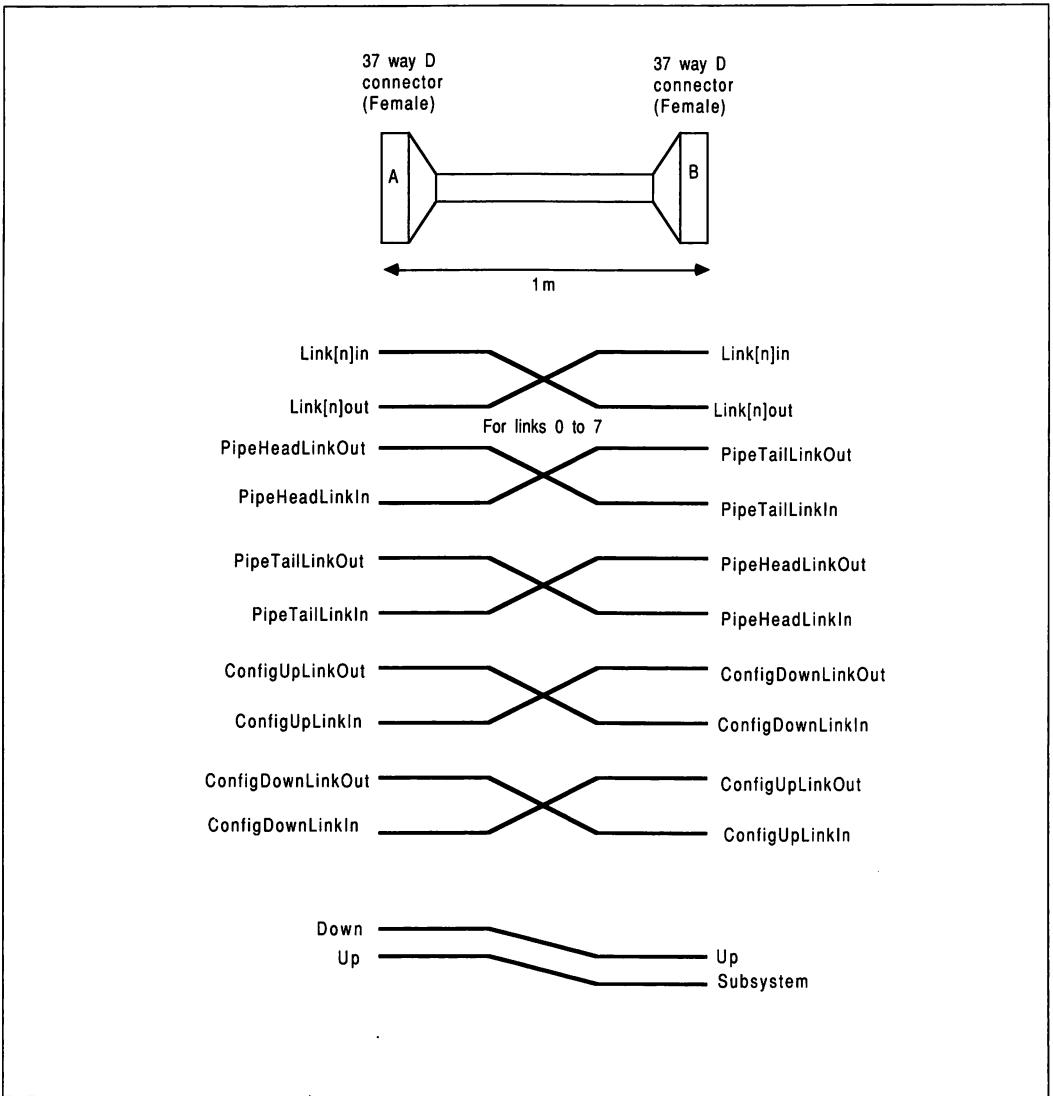
Figure 5.20 37-way cable

	c	b	a
1	GND	GND	GND
2	VCC	VCC	VCC
3	PAUX	nc	PAUX
4	VCC	VCC	VCC
5	GND	GND	GND
6	VCC	VCC	VCC
7	GND	GND	GND
8	nc	nc	nc
9	PipeHeadOut	Spare linkout	PipeTailOut
10	PipeHeadIn	Spare linkin	PipeTailIn
11	GND	GND	GND
12	nc	nc	nc
13	GND	GND	GND
14	nc	nc	nc
15	ConfigUpOut	Spare linkout	ConfigDownOut
16	ConfigUpIn	Spare linkin	ConfigDownIn
17	GND	GND	GND
18	nc	nc	nc
19	Spare	nc	Spare
20	Spare	nc	nc
21	Spare	GND	nc
22	Spare	nc	notSubReset
23	Spare	Spare linkout	notSubAnalyse
24	Spare	Spare linkin	notSubError
25	Spare	GND	GND
26	Spare	nc	nc
27	nc	GND	nc
28	notUpReset	nc	notDownReset
29	notUpAnalyse	Spare linkout	notDownAnalyse
30	notUpError	Spare linkin	notDownError
31	GND	GND	GND
32	GND	GND	GND

Table 5.1 P2 DIN 41612 connector pin out
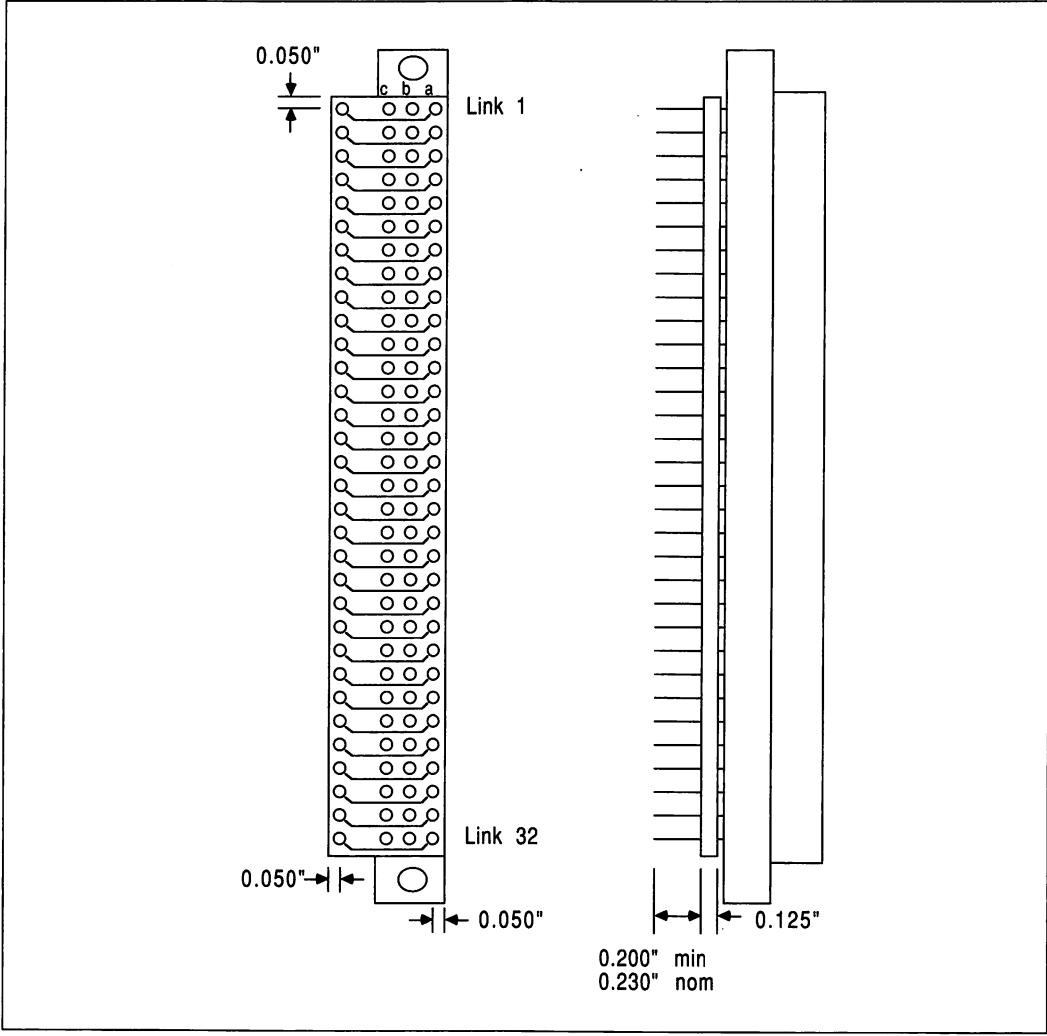
Figure 5.21 P1 32-link connector

	c	b	a
1	LinkOut0	LinkIn0	GND
2	LinkOut1	LinkIn1	GND
3	LinkOut2	LinkIn2	GND
4	LinkOut3	LinkIn3	GND
5	LinkOut4	LinkIn4	GND
6	LinkOut5	LinkIn5	GND
7	LinkOut6	LinkIn6	GND
8	LinkOut7	LinkIn7	GND
9	LinkOut8	LinkIn8	GND
10	LinkOut9	LinkIn9	GND
11	LinkOut10	LinkIn10	GND
12	LinkOut11	LinkIn11	GND
13	LinkOut12	LinkIn12	GND
14	LinkOut13	LinkIn13	GND
15	LinkOut14	LinkIn14	GND
16	LinkOut15	LinkIn15	GND
17	LinkOut16	LinkIn16	GND
18	LinkOut17	LinkIn17	GND
19	LinkOut18	LinkIn18	GND
20	LinkOut19	LinkIn19	GND
21	LinkOut20	LinkIn20	GND
22	LinkOut21	LinkIn21	GND
23	LinkOut22	LinkIn22	GND
24	LinkOut23	LinkIn23	GND
25	LinkOut24	LinkIn24	GND
26	LinkOut25	LinkIn25	GND
27	LinkOut26	LinkIn26	GND
28	LinkOut27	LinkIn27	GND
29	LinkOut28	LinkIn28	GND
30	LinkOut29	LinkIn29	GND
31	LinkOut30	LinkIn30	GND
32	LinkOut31	LinkIn31	GND

Table 5.2 P1 DIN 41612 connector pin out

6 Dual inline transputer modules (TRAMs)

6.1 Background

INMOS has built a number of transputer evaluation boards. Most are the same size (220mm x 233.4mm), which fits the INMOS ITEM. These boards have different transputer configurations and different amounts of memory (IMS T212, T414, T800, M212, transputer graphics, several transputers, 64K to 2M of RAM). INMOS has also produced boards to fit particular computers, such as the IBM PC and the NEC 9801.

The need

It would have been nice if we had been able to offer all the different transputer configurations to fit into these personal computers. But instead of about ten different designs of boards, this would have meant 30 different designs. And there was market demand for transputers to plug into VME, to VAX, to SUN, to other workstations, process control computers, minicomputers, mainframes. And there was further demand for more configurations, such as more memory per transputer, more transputers with less memory, or the same memory in much less space, graphics and other different peripherals......

Clearly to produce all these different transputer configurations, to plug into all these different computers, would need over 100 different board designs. Even if INMOS could design those, it would be foolish to stock and sell so many different designs. But a genuine market demand existed to be met. Somehow we had to separate the transputer configuration from the computer and its size and shape of board.

Meeting the need

A small range of transputer configurations, implemented as modular subsystems, and a small range of motherboards with sockets for the modules, offered this separation.

Users can mix and match different physical sizes of modules, modules with different memory sizes and modules with different functions. By mixing and matching, many more than 100 different combinations are possible.

An advantage to many customers who have the expertise in interfacing to their own computers is that they can design their own module motherboards, and use the ready-built transputer configuration supplied as modules. This should greatly reduce the time needed to prototype a transputer system.

The building block

In effect the module is a board level transputer, with a very simple standardized interface. The building block concept is practically realized by integrating memory and peripheral functions on board, and by limiting the pin out to 16 pins (although some modules use several sets of these 16 pins). It is just as easy to build transputer circuits with modules as it used to be to build logic circuits out of TTL.

Several of the modules are densely packed, offering thousands of MIPs, hundreds of MFLOPs and many megabytes, all on a few motherboards in a small box.

Two questions

Two questions are frequently asked - why DIL, and why just this size?

We use DIL because it is more robust than SIL when assembled on the board; also because the height of a transputer SIL strip would be over 1" using PGA transputers. The pin out of adjacent modules is arranged, however, so that if at some future time SIL strips appear viable, the SIL pinout works.

The size comes from considering how small a transputer could become. As the chip is about 1cm square, it would not fit with a 0.3" 16 pin DIP, but it would fit into a 0.6" 16 pin DIP. Put four of these on a regular prototyping board with rows of sockets on 0.3" centres and you have a set of pins 9-16 just 3.3" away from pins 1-8. Add enough at each end for mechanical fixing and width for a PGA to give the final size.

So the size was primarily chosen to fit standard prototyping boards. Conveniently, the size also fits the IBM PC, VME boards, and the INMOS ITEM, as well as a host of other computers.

6.2   Introduction

TRAMs are small subassemblies of transputers (or other components with INMOS links), a few discrete components, and sometimes some RAM and/or application specific circuitry. They:

> interface to each other via INMOS links
>
> have a standard pinout
>
> come in a range of standard sizes

The basic size of a TRAM is 1.05" by 3.66" overall, about half the size of a credit card. This basic size is referred to as Size1. Larger TRAMs can be up to 8.75" by 3.66", which fits comfortably on an IBM PC board or on a VME board (this largest size is referred to as Size8). Smaller TRAMs (hybrids or silicon, not yet implemented) can be as small as a 16 pin DIP with leads on 0.6" centres.

The standard pinout and standard sizes of TRAMs make it very simple for users to build customized mother-boards with sockets for TRAMs. These can either be in prototype form (Perfboard, Vectorboard or Veroboard), or in printed circuit form.

TRAMs may be plugged into the TRAM sockets on any of the following INMOS evaluation boards: B006 (eight Size1 modules), B009 (one Size4 module), B010 (four Size1 modules), and B011 (two Size1 modules). Connections between modules are hard wired on the B006 as two squares; on the other boards the links are connectable either at header plugs or at an edge connector.

The IMS B008 and B012 are specifically designed for TRAMs. Both boards can be connected into a wide variety of different networks by 'softwiring' connections between transputers by using the IMS C004 link switch. The B008 takes 10 Size1 TRAMs and plugs into the IBM PC, The B012 takes 16 Size1 TRAMs on a double extended Eurocard and plugs into the INMOS ITEM. INMOS will introduce other boards to fit other hosts.

The TRAM standards refered to above are independent of:

> transputer type (IMS T212, T414, T800, M212, etc.)
>
> number of transputers (1, 4, 8, 12, 16 are all possible)
>
> wordlength of transputer (16 bits on T212, 32 bits on T414)
>
> speed (T414-15, -20, to T800-30 and beyond)
>
> function (transputer plus RAM, disk control, other peripheral control)
>
> memory size (no external RAM up to many megabytes)
>
> package (68 pin PGA, 84 pin PGA, PLCC, and other transputer packages)
>
> implementation (through-hole PCB, surface mount PCB, hybrid, silicon)

Further information is available from INMOS on the B008 and B012 module motherboards, and on the product family of TRAMs.

6.3   Functional description

6.3.1   Pinout of size1 module

The pins include four INMOS links, which require no off-module buffering.

Table 1 shows the pinout. This pinout has been chosen partly to simplify layout of the motherboard, and partly to simplify the layout of the TRAM.

Table 1: Standard TRAM pinout

1	Link2out	Link3in	16
2	Link2in	Link3out	15
3	VCC	GND	14
4	Link1out	Link0in	13
5	Link1in	Link0out	12
6	LinkSpeedA	notError	11
7	LinkSpeedB	Reset	10
8	Clockin(5MHz)	Analyse	9

When LinkSpeedA and LinkSpeedB are both low, the TRAM links operate at 10Mbits/s. When they are both high, the links operate at 20Mbits/s. Other states of these pins are reserved for future enhancements.

The notError signal is driven by an open collector transistor so the signal can be wire ORred. This allows for the error line to be bussed in the same way as Clock, Reset, and Analyse. The fan-in of the notError signal must be controlled, and it is recommended that no more than ten notError outputs are wired together.

Pin 1 is marked by a silk screened triangle.

6.3.2   Pinout of larger sized modules

Figure 6.1 shows two adjacent Size1 TRAMs side by side. Notice that the orientation of the two modules is different. This difference in orientation serves two purposes: cooling of Size1 modules is improved; and it makes it possible at some future date to have Single-In-Line modules.
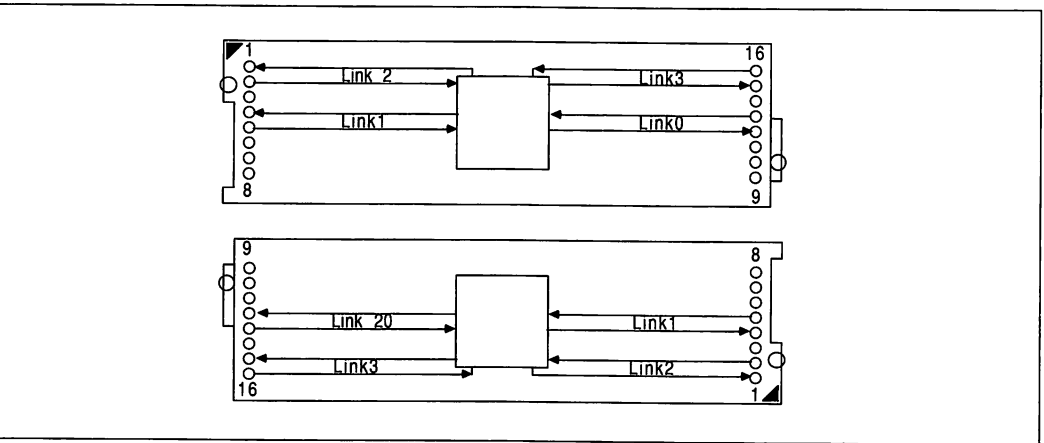
Figure 6.1 Orientation of adjacent Size1 modules

Many modules, and all the early products IMS B401 to B405, contain a single transputer, and so do not need more than one set of 16 pins for electrical signals. Modules larger than Size1, however, are assembled with extra sets of 16 pins; the extra pins give mechanical support, allow modules to be stacked, and provide extra GND and VCC pins. A Size2 module with one transputer is shown in figure 6.2a.

a

b
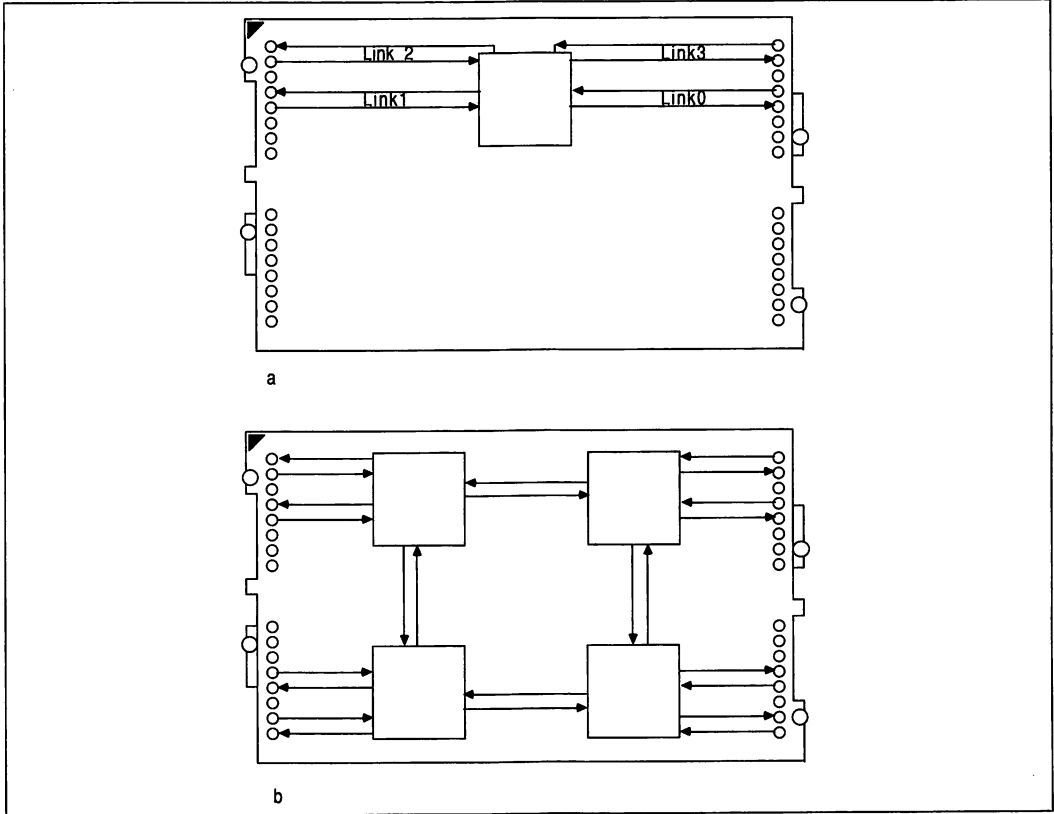
Figure 6.2 Size2 TRAMs with one and four transputers

TRAMs may be built with more than one transputer, or with transputers having more than four links. An example of a posible TRAM with more than one transputer is shown in figure 6.2b. This has four transputers connected as a square, in the same way as the IMS B003 and B006. (In practice, if INMOS were to produce a TRAM with four transputers, the links would probably be routed to make better use of standard motherboard connections.)

The detailed pinouts of larger modules are shown with the mechanical details in section 6.8 and assume that each TRAM has a single transputer, with four links.

Notice that the Size2 module and the Size4 module have the pins which are actually used at one end. The Size8 module (when it has a subsystem capability) has the pins which are used in the middle.

6.3.3   TRAMs with more than one transputer

Standards for pinout of transputers with more than one transputer are to be defined.

6.3.4   Extra pins

TRAMs may include application specific circuitry which requires pins other than the standard 16 pins. Examples are peripheral controllers or pipelines used for graphics or signal processing. The recommended connector for these is a strip of pins on 0.1" grid, such as a stripcable socket will attach to.

6.3.5   Subsystem signals driven from a TRAM

It is useful for TRAMs to be able to control a network· of transputers and/or more TRAMs. Such a slave network is known as a *subsystem* of the master, and the set of control signals from the module are described as a *subsystem port*.

The subsystem port consists of three signals: **SubsytemReset** and **SubsystemAnalyse**, which enable the master to reset and analyse its subsystem; and **SubsystemnotError**, which is used to monitor the state of the error flag in the subsystem. The polarity of these signals is such that a motherboard can be built with a master TRAM controlling slave TRAMs via its subsystem port with no buffering or gating. (Note that a change of polarity may be required for a subsystem port which goes off the motherboard.)

The three subsystem signals are located on low profile sockets which are positioned 0.1" inside the standard module pins 1-3. This is illustrated by figure 6.3.
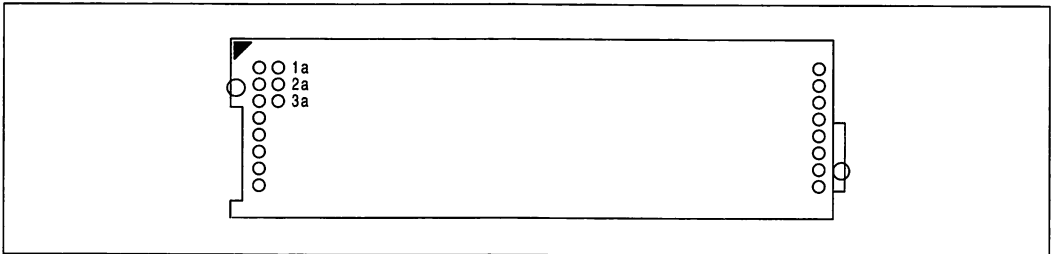
Figure 6.3 Location of subsystem sockets

The pinout is as follows:

Pin	Signal
1a	SubsystemnotError
2a	SubsystemReset
3a	SubsystemAnalyse

The sockets are fitted into the module PCB upside-down. The motherboard into which the module is plugged will also have three such sockets in the corresponding positions, but fitted from the component side in the usual fashion. The connection between the module and the motherboard is then made by a double-ended header, strip (see figure 6.4). This arrangement ensures that if the subsystem port of a module is not used, the module remains mechanically compatible with modules which do not have subsystem ports.
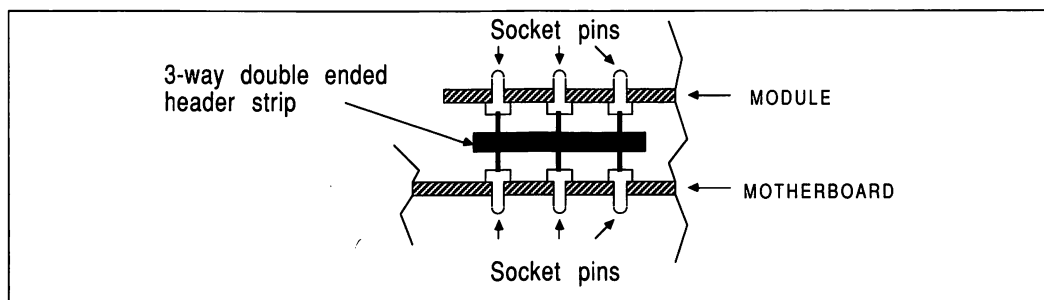
Figure 6.4 Subsystem port connections

Subsystem registers

The subsystem is controlled by reading and writing to addresses in positive address space (i.e. location zero onwards). On all INMOS evaluation boards and TRAMs, two BYTE locations are used, where each byte is the least significant byte of a 32 bit word. A further two locations control parity generation logic, which will be described in section 6.3.6. These four locations are permitted to repeat throughout the whole of the positive address space.

The subsystem registers are located at the following addresses for 32 bit transputers

Register	Hardware byte address
SubSystemResetLatch (write only)	#00000000
SubSystemAnalyseLatch (write only)	#00000004
SubSystemnotError (read only)	#00000000

The subsystem port operates as follows:

Writing a 1 into bit 0 of #80000000 asserts SUBSYSTEM Reset;
Writing a 0 into bit 0 of #80000000 deasserts SUBSYSTEM Reset.

Writing a 1 into bit 0 of #80000004 asserts SUBSYSTEM Analyse;
Writing a 0 into bit 0 of #80000004 deasserts SUBSYSTEM Analyse.

A 1 read from bit 0 of #80000000 indicates that SUBSYSTEM Error is TRUE.
A 0 read from bit 0 of #80000000 indicates that SUBSYSTEM Error is FALSE.

The subsystem is reset or analysed under the control of the transputer on the TRAM, but must also be reset when the TRAM itself is reset. To pass the signals on to the subsystem, the following combinational logic is included:

SubsystemReset = Reset OR SubsystemResetLatch
SubsystemAnalyse = Analyse OR SubsystemAnalyseLatch
the latches are initialized at power-on to be inactive.

Note that SubsystemError does NOT propagate to the TRAM's notError pin.

Multiple subsystems

TRAMs may contain more than one subsystem port. They should have their locations separated by 16 bytes.

6.3.6   Memory parity

TRAMs may include parity logic for external RAM. The implementation on TRAMs must ensure that there is no way that corrupt data can reach any other transputer.

One way to achieve this is that if a parity error occurs, the wait signal is held active so the memory cycle does not complete. All data in memory is lost, however, when an error occurs, and the memory cycle is slowed down by the parity check.

Parity checking may be enabled or disabled by writing to a parity control register. If parity is enabled and an error occurs, the error is ORed in to the notError signal from the module. Information on the cause of the error can be found by examining the parity status register.

Reset disables parity checking and deasserts MemWait. When the transputer is analysed, MemWait is deasserted and the contents of the parity status register are preserved.

The parity registers are as follows:

Register	Hardware byte address
Parity control (write only)	#00000008
Parity status (read only)	#00000008

The locations are used as described below:

Writing a 1 into bit 0 of #80000008 enables parity error detection;
Writing a 0 into bit 0 of #80000008 disables parity.

Reading the contents #80000008 returns the status of the parity detection hardware.

Bit	Status
Bit 0	Indicates a parity error has occured.
Bits 1 & 2	Indicate the BYTE in which the error occured. (Bit 1 is lsb).
Bits 3..n	Indicate the BANK in which the error occured. (Bit 3 is lsb).

6.3.7   Memory map

The memory map should be of the form:

ROM	top of memory
Peripherals	
Subsystems	
External RAM	
On-chip RAM	bottom of memory

In the particular case of TRAMs with 32 bit transputers, the memory map should be as follows:

Byte address	Description	Comment
7FFF FFFF		Bootstrap program requires ROM at top of memory.
7FFF FFFE	Boot from ROM	7FFF FFFE will contain a backward jump to the bootstrap.
	Peripherals	If used
0000 000C		These locations must
0000 0008	Parity status and control	be decoded as a set
0000 0004	SubsystemAnalyseLatch	of four, even if Parity
0000 0000	SubsystemResetLatch	is not used.
8FFF FFFF	RAM	Both internal and
Memstart	RAM	external RAM

Substantial logic can often be saved by not fully decoding the hardware address. An effect of not fully decoding the address is that hardware can appear at multiple addresses.

In particular, if the module does not have a subsystem, the RAM can repeat throughout the address space, including the positive address space (above location 0).

The Subsystem and parity locations can also repeat throughout the positive address space.
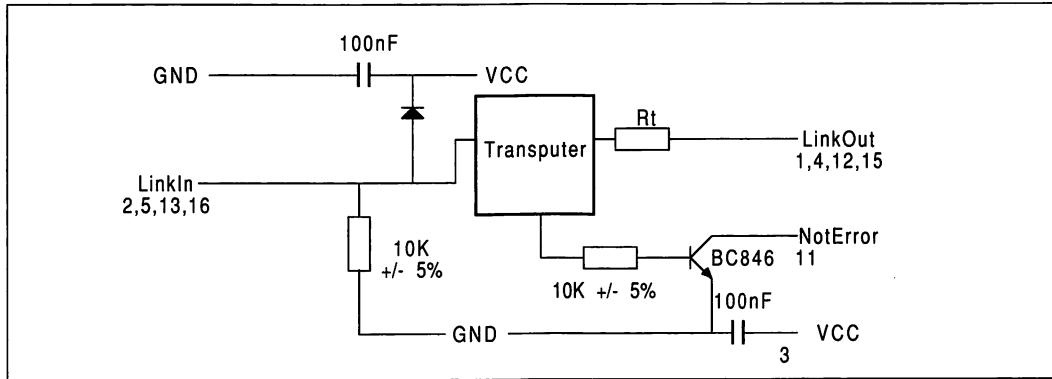
Figure 6.5 Recommended circuit between TRAM pins and transputer

6.4   Electrical description

6.4.1   Link outputs

Link outputs must be terminated so that the combined output impedence of the transputer plus termination resistors is 100 ohms ± 20%. For the optimum value of resistor, see the appropriate transputer data sheet.

6.4.2   Link inputs

Link inputs may be taken off a module motherboard and so must be protected from positive ESD by a diode to VCC. Signal diodes such as 1N4148 or LL4148 may be used. To prevent an unconnected link input from floating high, link inputs must be pulled down to GND by a resistor, preferred value 10K ± 5%.

6.4.3   notError output

The notError output is a wired OR signal driven by an open collector or an open drain. Maximum leakage should not exceed 10 microamps. Maximum saturation voltage when the transistor is ON and is sinking 10 mA should not exceed 0.4 V. A suitable transistor is BC846 (SOT23) with a 10K resistor between the transputer's Error pin and the transistor base. The pullup resistor on the module motherboard should draw between 5mA and 10mA when a transistor is ON.

Although the above is conservative and should allow a fan-in of several hundred, it is recommended that the fan-in is limited to 10.

6.4.4   Reset and analyse inputs

These signals are connected directly from the TRAM pins to the transputer. They must always be driven by buffers on the module motherboard. Because the motherboard will often have filters on the Reset and Analyse signals, the Reset pulse width should be much wider than specified for the transputer. Recommended pulse width is 5 ms, with a delay of 5 ms before sending anything down a link.

6.4.5   Clock input

The TRAM must not present excessive capacitance to the clock input signal. The clock input should therefore be limited to a single load, which should be connected to the TRAM pin by a trace no longer than 30mm.

Particular care should be taken on the module motherboard to ensure that the clock input is clean, with fast edges, minimal undershoot, and minimal jitter (see transputer data sheet for clock specification).

6.4.6   notError input to subsystem

The notError input should not have a pullup resistor on the TRAM. The pullup resistor must be on the motherboard.

6.4.7   GND, VCC

Adequate high frequency decoupling capacitors must be used. In particular there should be decoupling capacitors close to the GND pin and to the VCC pin of each TRAM. Recommended value is 100 nF, preferably at least half as many as the module has ICs.

6.5   Mechanical description

In the following, dimensions are quoted in inches for PCB length, width and related dimensions; all other dimensions are quoted in millimetres.

6.5.1   Width and length

TRAM module size	A	B	C	D
1	3.3"	3.5"	1.05	3.66"
2	3.3"	3.5"	2.15"	3.66"
4	3.3"	3.5"	4.35"	3.66"
8	3.3"	3.5"	8.75"	3.66"
1/2	1.5"	1.7"	1.05"	
1/4	0.6"	0.8"	1.05"	
2/4	0.6"	0.8"	2.15"	

A (pitch between pins)
B (to inside edges of board)
D (overall, including fixing/polarizing lugs)
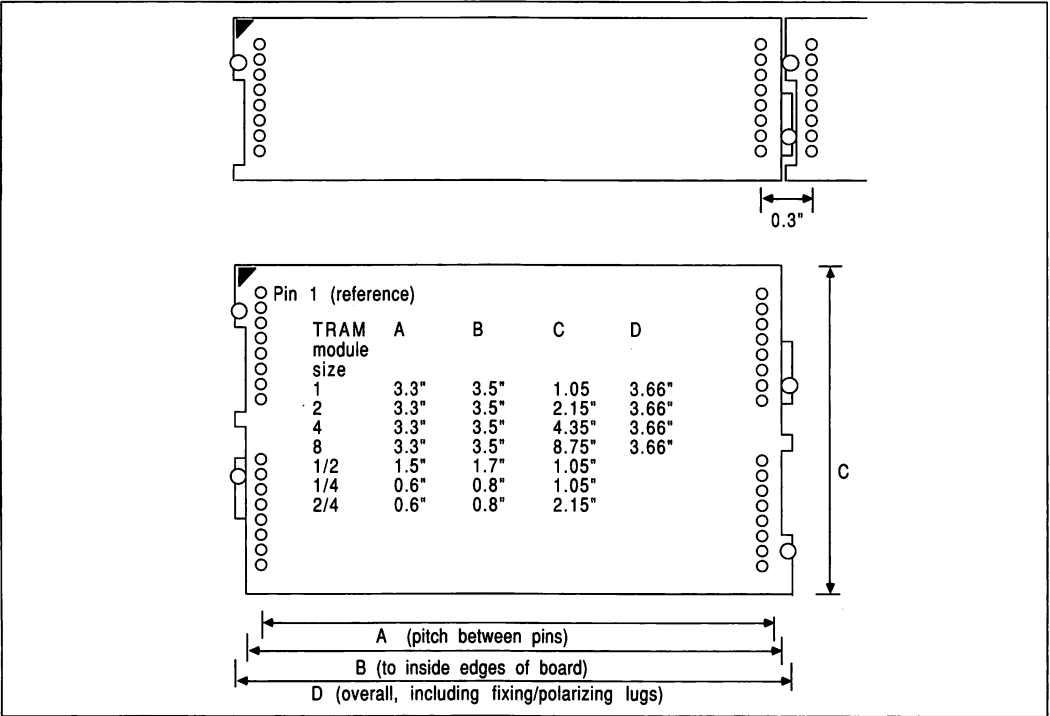
Figure 6.6 TRAM sizes

The basic size of a TRAM is a very wide 16 pin DIP, with 3.3" between the two rows of pins. These TRAMs fit on a 3.6" pitch on their length, and a 1.1" pitch on their width. Extra length is added beyond the pins to hold the pins, to provide for mechanical fixing, and to polarise the module shape.

TRAMs can be made larger than the standard size by keeping the 3.3" between pins and using two or more sets of the 16 pins.

TRAMs can be made smaller than the standard size, down to a 16 pin DIP with 0.6" between the two rows of pins, or 1.5" between the pins. These sizes will normally be used for single chip modules or hybrids.

In general the printed circuit TRAMs are longer than the pitch between the two rows of pins. The TRAMs are also wider than the 0.8" suggested by 16 pins. The small TRAMs may be side-brazed DIPs, as short as 0.8" long.

The top drawing in figure 6.6 shows a Size1 module and how the jigsaw pattern fits together between adjacent modules. The lower drawing in figure 6.6 shows the various sizes of TRAM. Detailed dimensions of the different sizes are given in section 6.8.

6.5.2   Vertical dimensions

There are no vertical height constraints for TRAMs. However, keeping the height of a TRAM, both below and above the board, within certain limits allows the TRAM to fit together with other TRAMs and motherboards.

Figure 6.7a shows height specifications which allow double-stacking of the TRAMs and which will allow two-deep stacked TRAMs on a motherboard to fit into a 1.0" pitch card-cage, (see figure 6.7e). Figure 6.7b shows how this vertical size fits onto a motherboard which has no components under the TRAM. Figure 6.7c shows the same TRAM fitted above components on a motherboard, using spacer socket strips to gain extra height.

Figure 6.7d shows another height specification which allows components such as zip packaged ICs and SMB connectors to be used on the TRAM, whilst permitting these TRAMs to fit onto motherboards in a 0.8" pitch card cage. Note that this is only possible when there are no components under the TRAM on the motherboard.

It is recommended that any component reaching a maximum specified height has an insulating surface.

Note that the datum for component heights on both sides of the TRAM is the component side surface. This datum is also used for the stackable socket to minimize tolerance buildup.
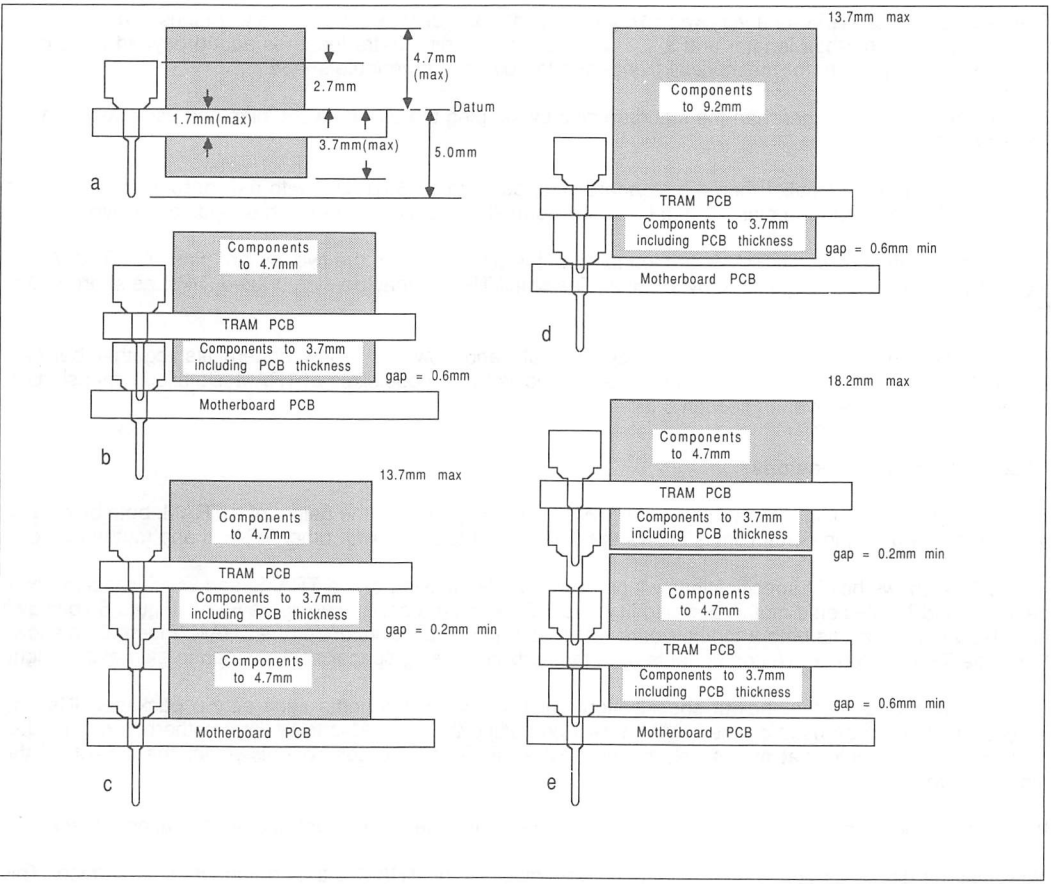
Figure 6.7 Component heights

Components must not interfere with the TRAM pins, and so the area shown in figure 6.8 must be left free of components.
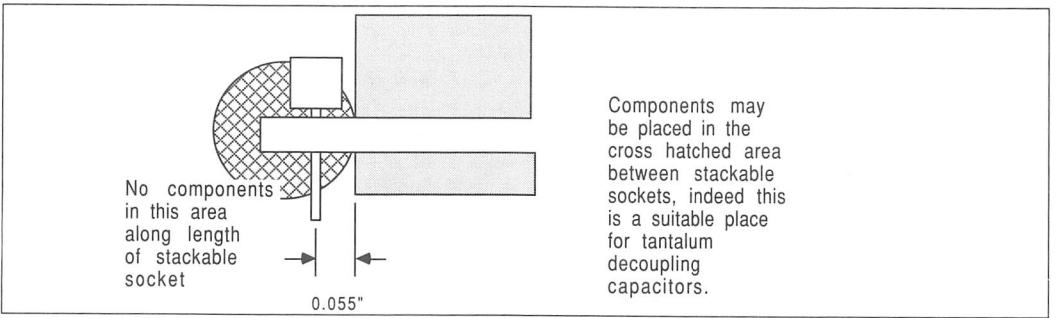
Figure 6.8 Area close to TRAM pins

6.5.3   Direction of cooling

TRAMs should be designed so that cooling air can flow freely across the width of the the the module, or in other words parallel to pins 1 to 8 rather than from pin 1 to pin 16. Care should also be taken to ensure that the surface of a module is not too flat: projections cause turbulence which improves cooling.

6.6   TRAM pins and sockets

6.6.1   Stackable socket pin

The stackable pin socket is shown in figure 6.9.
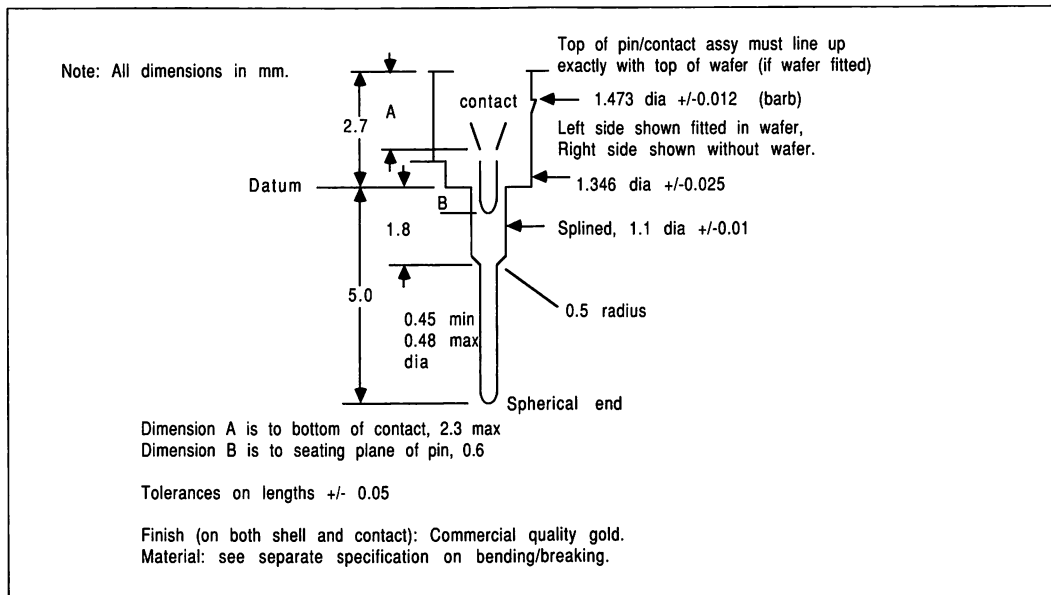
Figure 6.9 Stackable socket pin

Approved manufacturers of the stackable socket pin are (with part numbers): [1]

	Individual socket pin	**Strip of 8 sockets**
Scott	128-446	15108-128-446

The individual socket is used on the TRAMs themseleves. Strips of 8 sockets are used on TRAM mother-boards and as spacers (as in figure 6.8) between TRAMs and motherboards.

6.6.2   Through-board sockets

The component height given in figure 6.7 means that there is not enough height for conventional sockets for the components. A number of manufacturers make sockets which fit into a PCB in such a way that the thickness of the PCB is used for the socket, rather than extra height above the board.

---

[1] These parts are available from Scott Electronics Ltd, Tonbridge, Kent, England (Tel: 0372 359270), or Andon Electronics Corp, Albion, RI, USA (Tel: 401 333 0388).

INMOS has seen and used the following sockets. No particular recommendation for any of these is given or implied. Other manufacturers have shown data sheets for similar sockets with a height of approximately 0.8mm. The Augat 'Holtite' sockets, which sit below the PCB surface, have been seen but not used. The Augat 'Soldertite' sockets have similar dimensions to the Harwin 3153 and have been seen in prototype quantities. All of the sockets are available individually or assembled into strips; some are available in DIP and PGA format.

Manufacturer	type	height above PCB
Harwin (UK)	H 3153-01	0.38mm
Mark Eyelet (AMP) (US)	M8043PEC	0.2mm approx
PreciDIP (Switzerland)	014-92-001-41-012	0.4mm
Advanced Interconnections (US)	type -85	0.78mm
Harwin (UK)	H 3155-01	1.2mm
PreciDIP (Switzerland)	type 1407	0.8mm

6.6.3   Subsystem pins and sockets

The preferred socket to fit on the solder side of the TRAM is Harwin H 3153-01, and on the motherboard also. Samtec pin strip HLT-03-G-R is suitable for connecting between these sockets.

6.6.4   Motherboard sockets

The TRAM pins/stackable sockets will plug into any standard IC socket. To meet the component heights given in figure 6.7, the stackable socket (see section 6.6.1) must also be used on the motherboard.

Motherboard sockets for the Subsystem signals should be the 0.38mm or 0.4mm sockets referred to above.

6.7   Mechanical retention of TRAMs

Vibration tests have shown that in a normal office or laboratory environment, the TRAMs remain plugged into their sockets. In transit, however, or in an environment where there is vibration, some form of mechanical retention may be necessary.
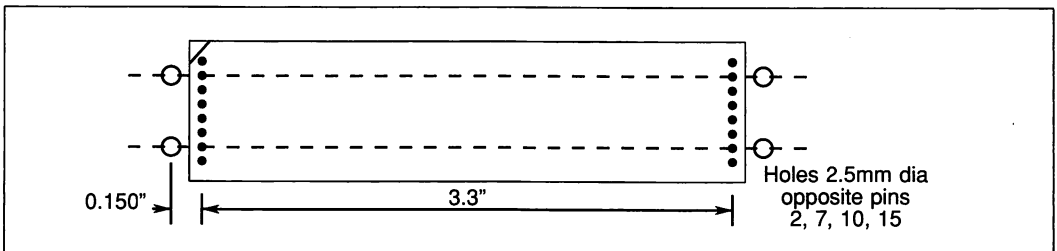
Figure 6.10 Fixing holes for mechanical retention

The detail drawings of the module sizes in section 6.8 show fixing holes in the modules. Similar fixing holes should be drilled in the motherboard as shown in figure 6.10. M2.5 nylon bolts may be used between these fixing holes to secure the modules.

6.8   Profile drawings
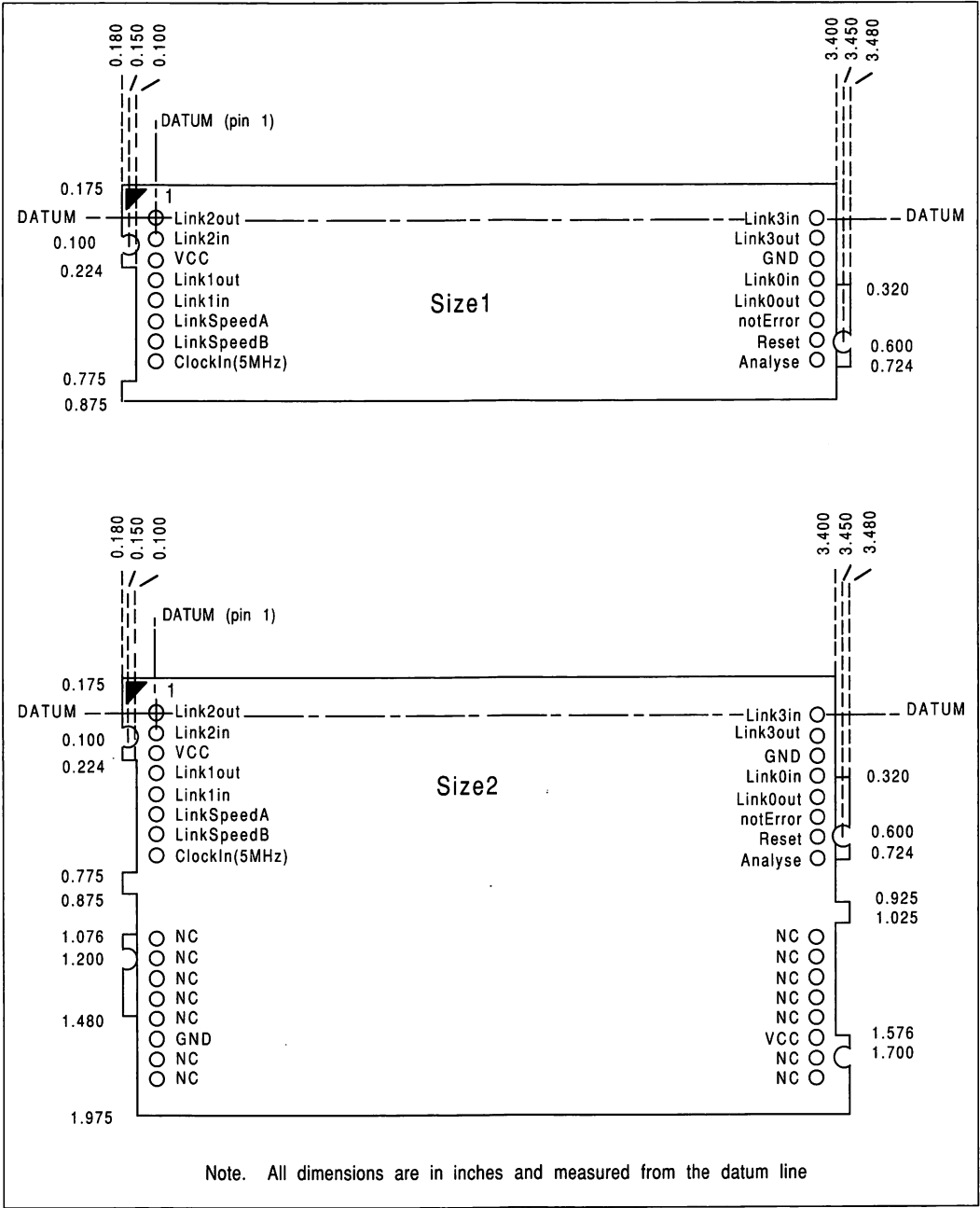
Figure 6.11 PCB profile drawings and pinout, TRAMs Sizes 1 and 2

0.180
0.150
0.100

3.400
3.450
3.480

DATUM (pin 1)

0.175

DATUM —

0.100

0.224

○ Link2out — — — — — — — — — — — — — Link3in ○ — — DATUM

○ Link2in   Link3out ○

○ VCC   GND ○

○ Link1out   Link0in ○   0.320

○ Link1in   Link0out ○

○ LinkSpeedA   Size4   notError ○

○ LinkSpeedB   Reset ○   0.600

○ ClockIn(5MHz)   Analyse ○   0.724

0.775

0.875

0.925

1.025

1.076   ○ NC   NC ○

1.200   ○ NC   NC ○

        ○ NC   NC ○

        ○ NC   NC ○

1.480   ○ NC   NC ○

        ○ GND   VCC ○   1.576

        ○ NC   NC ○   1.700

        ○ NC   NC ○

                                                               1.975

2.025

        ○ NC   NC ○

2.300   ○ NC   NC ○

2.424   ○ VCC   GND ○

        ○ NC   NC ○   2.520

        ○ NC   NC ○

        ○ NC   NC ○

        ○ NC   NC ○   2.800

        ○ NC   NC ○   2.924

2.975

3.075

3.125

3.225

3.276   ○ NC   NC ○

3.400   ○ NC   NC ○

        ○ NC   NC ○

        ○ NC   NC ○

3.680   ○ NC   NC ○

        ○ GND   VCC ○   3.776

        ○ NC   NC ○   3.900

        ○ NC   NC ○

4.175

Note.   All dimensions are in inches and measured from the datum line

Figure 6.12 PCB profile drawings and pinout, TRAMs Size 4

Size8 module
without subsystem

DATUM (pin 1)

Link3in
Link3out
Link0in
Link0out
notError
Reset
Analyse

Link2out
Link2in
VCC
Link1out
Link1in
LinkSpeedA
LinkSpeedB
ClockIn(5MHz)

Figure 6.13 PCB profile drawing and pinout, TRAMs Size8 without subsystem
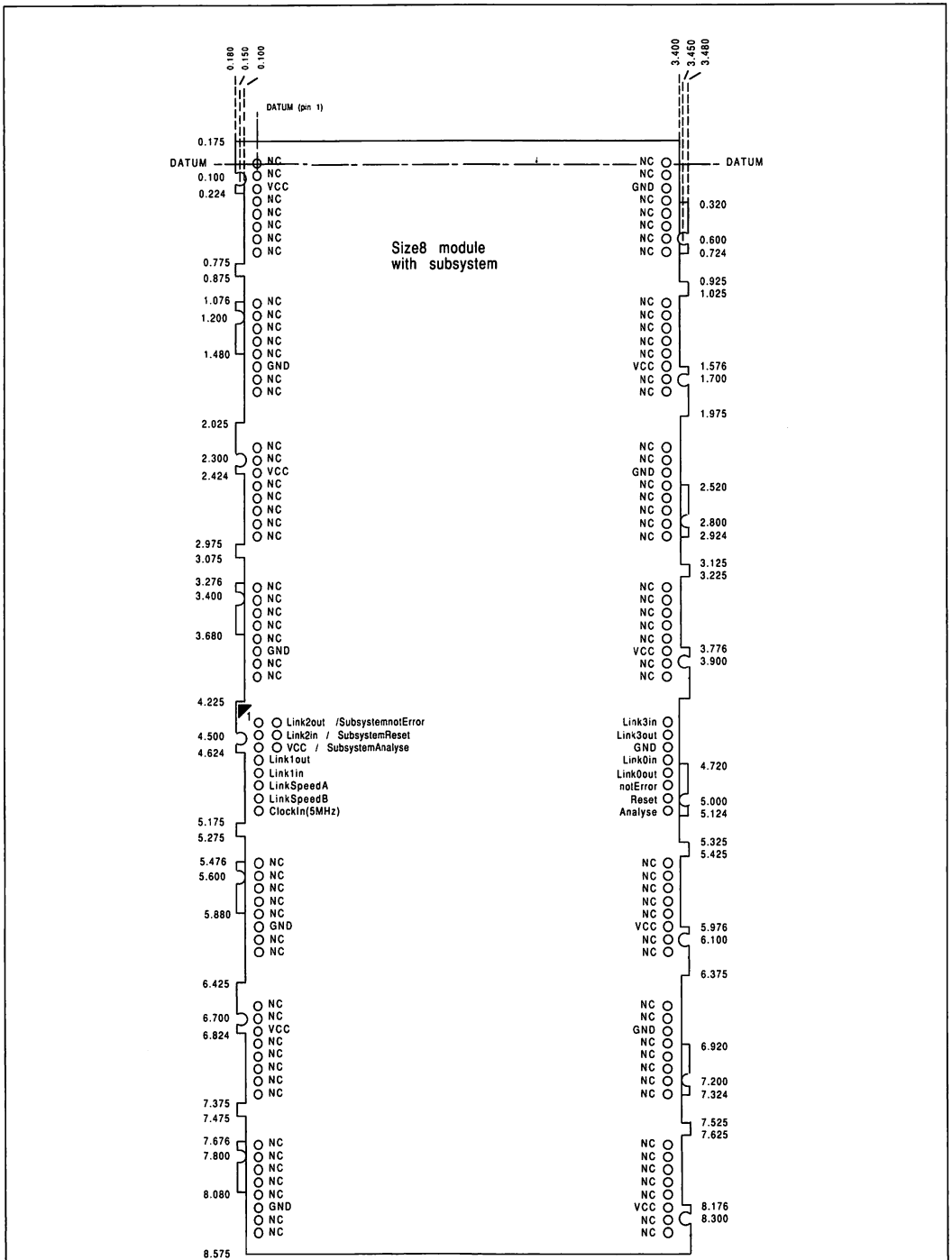
Figure 6.14 PCB profile drawing and pinout, TRAMs Size8 with subsystem

Software

7 Program design for concurrent systems

7.1 Introduction

This note illustrates one approach to programming concurrent systems in occam. It concentrates on applications, rather than general purpose computer networks, which are covered in Technical Note 13 [1].

7.2 Structuring the system

There is no absolutely correct topology for an application; each possibility represents a trade-off between programming ease and ultimate efficiency. In this trade off consideration must be given to the level of reliability required and the cost of development and final hardware.

Assuming there is to be more than one processor in the system under design; an important early decision is the manner of sharing the load between the processors. This depends upon how the problem may be divided, and the measure of performance required. If the task is a repetitive one; that is, the same operation performed on many pieces of data, the ultimate throughput is infinite, limited only by economic factors; the number of processors you can afford. However, the latency; that is, the delay from raw data in to associated results out, cannot be reduced below the total execution time of those operations that must be performed sequentially on the data.

Having established that a task is divisible in the way we require, processes can be written to perform each subtask, and each data item passed through the subtasks. Whether divisible or not, the option of providing multiple processes; each capable of performing the same task, remains. This approach allows many items of data to pass through many identical processes at the same time and thus increases overall throughput.

Note that we use the term 'processes' in preference to 'processors'. The first term is the logical division of a task and the second is the physical division of a task. In the final analysis we may allocate several processes to one processor. This is an important point; as it illustrates that the division of a task into sub tasks must be done to a greater extent rather than a lesser, as processes can be grouped later, but cannot easily be subdivided after writing.

7.3 System topology

We can now consider the topology of the system. Processes are represented by rounded boxes, and communication channels by arrowed lines. To illustrate a simple case, consider the example in figure 7.1.
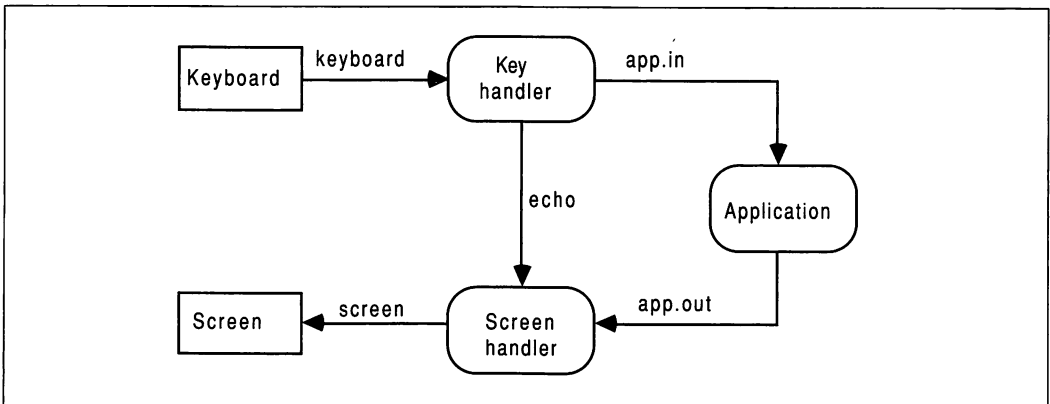
Figure 7.1

This shows a functional division of a generic application into a keyboard handler, a screen handler and the application itself. Such a division is for ease of programming and flexibility rather than performance.

Each channel is given a name on the diagram, and then the top level occam can be written. The three functional blocks execute at the same time, i.e. in PARallel. The ONLY items they share are the channels between them, so these are declared in an outer scope.

```
... proc decls
CHAN OF INT app.in, echo, app.out
PAR
    keyboard.handler (keyboard, app.in echo)
    screen.handler   (screen, app.out, echo)
    application       (app.in. app.out)
```

This top level design done - and instantly coded due to the correlation between the occam and the diagram - we progress to the three functional blocks.

These are totally independent, and as long as they agree on the form of data to pass between them, can be designed by different people on different sites. This hierarchical approach means that the most complex task can be attacked and reduced to simplicity.

The last example illustrated functional division. This is the most effective solution for ease of programming, but relies on a divisible task. For the indivisible task, the solution is 'many hands make light work' — achieved by distributing data items to different processors, all working at the same time. In the first example, the system topology was dictated by the connectivity required by the functions. In the indivisible task, the topology is arbitrary.

A simple topology directly supported by occam's PAR replicator syntax is a pipeline, or spaceline. The pipeline relies on each stage not only processing, but also passing on data and/or results on behalf of other processes.
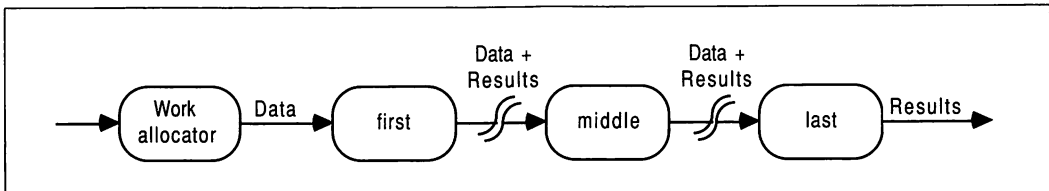
Figure 7.2

In order to achieve this, messages would have tags indicating their types and a router process would handle this, so each stage would become:
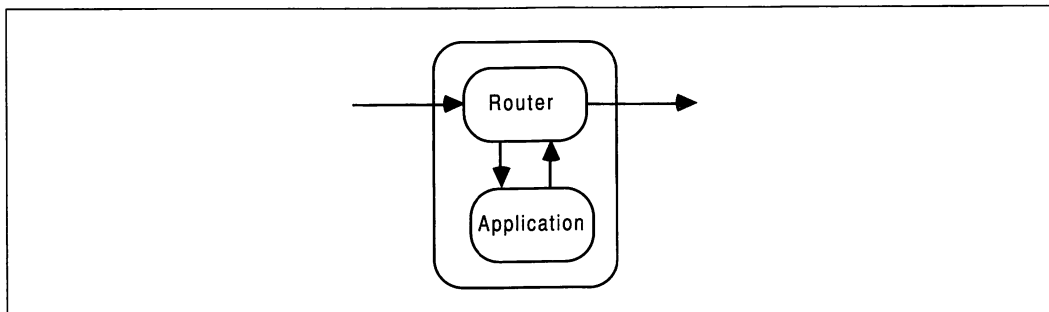
Figure 7.3

However, as channels are available in the opposite direction, one can arrange for input and output to be at one end of the pipeline, which allows for simple extensibility.
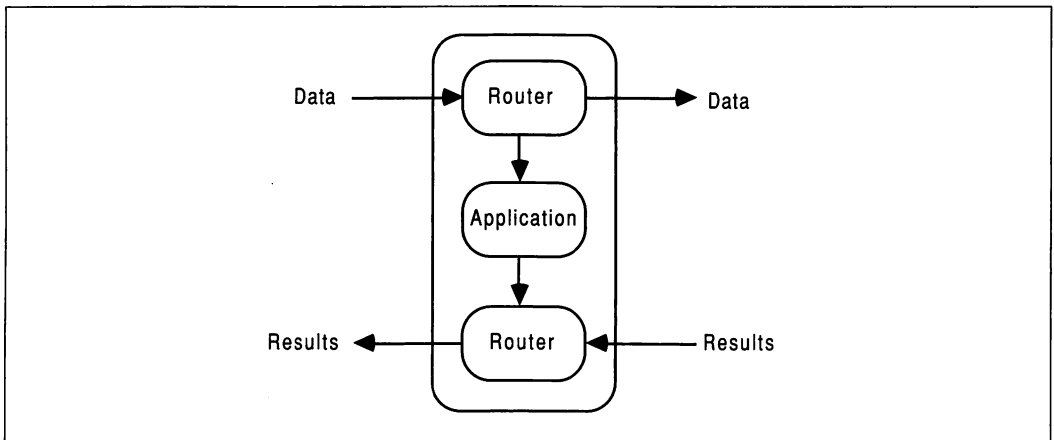
Figure 7.4

The routers are very simple — usually around 5 lines of operational code after initialisation etc., so are not a problem. However, it must be borne in mind that the first processors will be handling the data and results for ALL processors, so one must consider the balance of communications and processing. Provided messages are used, rather than single words or bytes, a pipeline is appropriate to length of order 10 (i.e. < 100)

A spaceline system is implemented as shown:
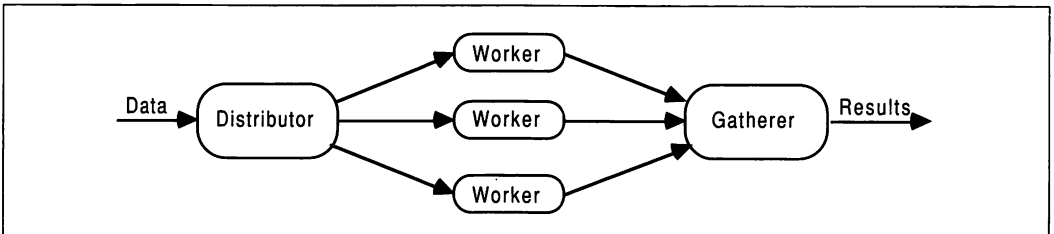
Figure 7.5

The width of a spaceline is limited by the number of links on the distributor and gatherer. By using a tree structure, spacelines of any width can be constructed.
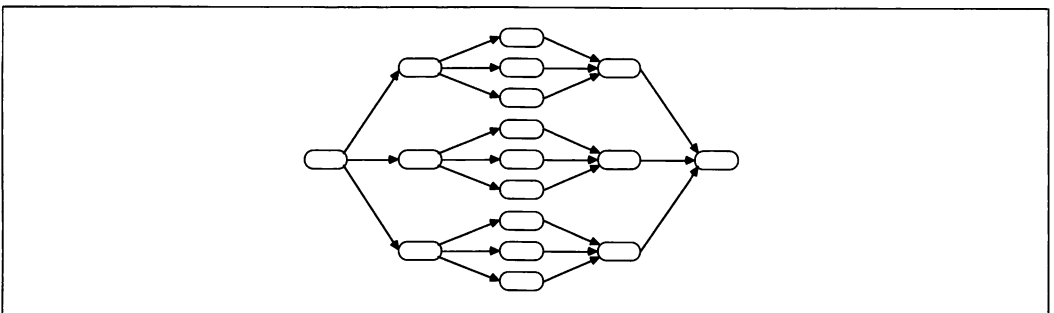
Figure 7.6

Clearly, the optimum topology is application dependent, and each application must be judged on its merits. The rest of this note will concentrate on functionally divided applications. For arrays etc. (See Technical Note 13 [1].)

7.4   System design – the functional blocks

Reverting to the example of figure 7.1, we must now design the functional blocks.

In general, each process must do some initialisation, then will repetitively receive data, and act upon it. The actions may be complex, may read more data, may generate output, and may terminate the process, but the basic structure still holds.

The Transputer Development System uses a folding editor, which can represent a large block of text in a single named fold line marked by three dots. A fold can contain another fold, nested to any depth. Folds can be 'opened' by the editor to display internal structure and source text, or 'closed' to hide data not currently of interest. Thus any level of detail can be viewed at will.

Folds can be created and named even before their contents have been written. This allows the structure of the process to be entered as part of the design. Thus the generic process is as shown here:

```
PROC my.proc (parameters)
  ...   declarations, including local procs
  SEQ
    ... initialisation
    WHILE condition
      SEQ
        ... loop initialise
        ... input data
        ... act upon it
        ... tidy up this pass
    ... tidy up process
:
```

Considerable experience training programmers new to both the folding editor and occam has shown that adopting this type of structure is essential, otherwise they immediately enter a program that mimics languages they are accustomed to, rather than making use of the parallel and communications of occam.

Thus the keyboard handler from the example becomes:

```
PROC keyboard.handler(CHAN OF INT in , out , to.screen)
  INT ch:                        --declarations
  VAL stopch IS INT '@':
  BOOL running:

  SEQ
    running := TRUE              --initialisation

    WHILE running
      SEQ
        in ? ch                  --input

        PAR                      --action
          out ! ch
          to.screen ! ch

        IF
          ch = stopch
            running = FALSE
          TRUE
            SKIP
:
```

As can be seen, many of the elements of the standard structure are null, but the conscious decision to exclude them is very beneficial in the design process.

One powerful construct of occam that does not clearly fit this structure is the ALTernate. This is used to take input from one of many channels, when it is not known which will be ready first. Thus it is used in the screen handler. The reason it does not clearly fit the standard format is because it includes both input and action. The screen handler implemented here puts echoed text and output text in two separate windows, so the structure is modified to:

```
WHILE <condition>
  ALT
    ... input from echo
      SEQ
        ... go to echo cursor position
        ... output text
        ... update cursor position

    ... input from application
      SEQ
        ... go to application cursor position
        ... output text
        ... update cursor position
```

Again the editor helps, because due to the similarity between the two branches, only one need be entered, it can then be copied and edited.

7.5   System integration

Once all three function blocks are entered, the system can be compiled and tested. Were it a complex application, the individual processes would have been separately tested, with test-data-generators, as per Technical Note 2 [2]. This example, however is simple enough that the complete system can be tested together.The modus operandi is first to run the program on a single transputer, either the development system or an external evaluation board, and then to adapt it for the target system.To adapt this program to run on 3 transputers is mechanical — one simply exchanges the PAR for a PLACED PAR, add PROCESSOR statements, assign the channel names to particular links using PLACE...AT, and make each PROC separately compiled.

```
...SC keyboard.handler
...SC screen.handler
...SC application

CHAN OF INT  keyboard,screen,echo, app.in, app.out:

PLACED PAR
  PROCESSOR 0 T4
    PLACE keyboard AT link0in:
    PLACE echo      AT link1out:
    PLACE app.in  AT link2out:

    keyboard.handler ( keyboard , app.in , echo )

  PROCESSOR 1 T4
    PLACE screen    AT link0out:
    PLACE echo      AT link1in:
    PLACE app.out  AT link2in:

    screen.handler ( screen , app.out , echo )

  PROCESSOR 2 T4
    PLACE app.in    AT link0in:
    PLACE app.out  AT link1out:

    application ( app.in , app.out)
```

However, in a more general system, if my advice was heeded, there are more logical processes than physical processors. The allocation must be done by the programmer considering three factors:

1 The connectivity — taking account of the number of physical links on each transputer.

2 The processor loading — the system will probably run at the speed of the most loaded processor.

3 The size of program on each processor, with regard to both internal memory (which is faster) and total memory provided.

Once the decision is taken, it is simply an additional box drawn on the diagram to map our example onto 2 processors.

In this case there is a little juggling to be done to ensure that the code for each processor is a single separately compiled unit.

```
...SC keyboard.and.screen.handler
...SC application

CHAN OF INT  keyboard,screen, app.in, app.out:

PLACED PAR
  PROCESSOR 0 T4
    PLACE keyboard AT link0in:
    PLACE screen   AT link0out:
    PLACE app.in   AT link1out:
    PLACE app.out  AT link1in:

    keyboard.and.screen.handler (keyboard,screen,app.in,app.out)

  PROCESSOR 1 T4
    PLACE app.in   AT link0in:
    PLACE app.out  AT link0out:

    application ( app.in , app.out)
```

For the multi transputer system, an additional operation is performed after the compilation known as configuring. This creates a code file that can be loaded into a network of transputers. It includes the routing information for the code, derived from the PROCESSOR and PLACE AT statements. The target system can then be loaded with a single keystroke, and live testing can begin — the multi processor concurrent program is running.

7.6   Conclusions

Concurrent programming is very simple, and errors easily avoided, using occam, provided the programmer is willing to adapt his style appropriately. Specification, design and programming become a smooth flow of work using the same tools on the same text, which becomes progressively more detailed. The process and channel diagram is essential in top down design, and at the lower levels, a formalised approach to design, using the folds where a COBOL programmer might have used flow charts allows on-screen design and rapid, error free programming.

7.7   References

1 *Transputer networks using the IMS B003*,
   Technical note 13, INMOS Limited, 1987.

2 *The transputer based navigation system – testing embedded systems*,
   Technical note 02, INMOS Limited, 1987

8 Exploring multiple transputer arrays

8.1 Introduction

A transputer is a component computing device which can easily be connected to form networks in multiprocessor arrays. These arrays can become quite large and complex. This technical note describes an 'exploratory worm program', which will explore an unknown network of transputers, and determine its configuration. This is useful in confirming that the transputers have been connected in a particular configuration, as required for some particular task, and that they are all working properly. Further applications include testing a network for reliability, and loading code into a network whose configuration is not known in advance.

The exploration is achieved by having a program which will worm its way around the network, exploring all the links on all the transputers to determine the interconnections. An example of an exploratory worm program, which is referred to in this technical note, is available as part of the Transputer Development System. This program explores a network made up of an unlimited number of IMS T414 transputers. Some notes about further applications are given in section 8.6.

8.2 The structure of an exploratory worm program under the TDS

The transputer development system (TDS) recognises two different types of program, known as **EXE** and as **PROGRAM**. An **EXE** program runs on the host transputer, and may access the keyboard, screen, and filing system of the host machine. A **PROGRAM**, on the other hand, runs on a network of one or more transputers, and is loaded from the host transputer via a transputer link. This link may be the network's only connection with the outside world.

An example of such a system is given in figure 8.1. This shows an IBM PC-AT with an INMOS B004 evaluation board, running a single IMS T414 transputer and 2 megabytes of external RAM. This transputer acts as the host processor for the development of programs, and for loading multiple transputer networks. Link 2 of the B004 is connected to an INMOS B003 evaluation board, which runs 4 IMS T414s, each with 256 kilobytes of memory.
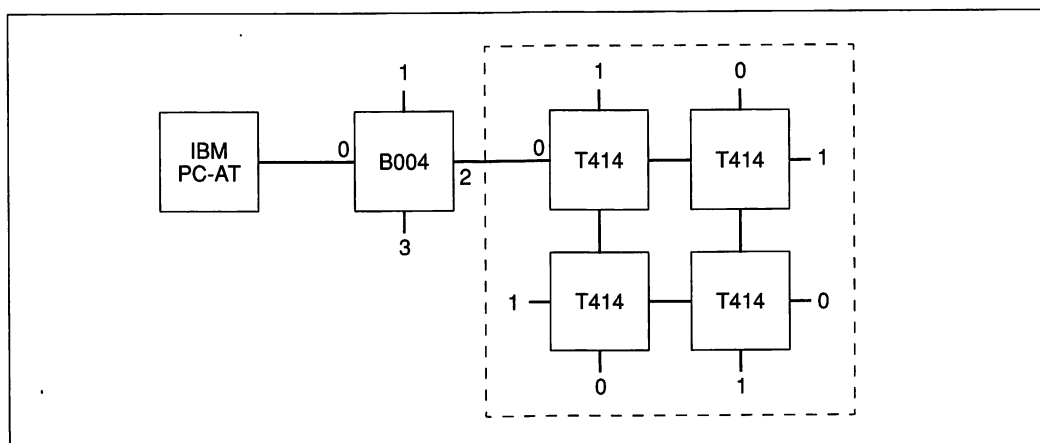
Figure 8.1

Typically, when a **PROGRAM** is loaded onto a multiple transputer network, a simple **EXE** program will also be run on the host transputer which monitors the output transmitted back from the **PROGRAM**, sends results to the screen, passes on any input from the keyboard, and controls the TDS filing system, as required.

A simple **PROGRAM**, intended to run on a network of just one transputer, looks like this:

```
{{{   PROGRAM Example
{{{F
...   SC Example
PROCESSOR 0 T4
  Example ()
}}}
}}}
```

When this bundle is compiled, configured and extracted, a new fold is created:

```
...F CODE PROGRAM Example
```

If extracted as a **BOOTABLE** type fold (as opposed to a **DIAGNOSTIC** fold), this **CODE PROGRAM** fold will just contain code which will initialise and load a single transputer, and run **SC Example**. Thus, if an occam byte array **Program** contains the contents of a bootable **CODE PROGRAM** fold, then the effect of:

```
ToLink ! Program
```

is to load and run the program on a transputer connected to link **ToLink**. The precise way in which a transputer loads code does not concern us here — it is described in full in [1].

A program may thus explore a network of transputers as follows:

Suppose that a transputer is already running an exploratory worm program, and that it is connected to another transputer, which has not yet been loaded with code. The first transputer, which will be called the 'parent', loads the second ('daughter') by outputting the code **Program** as above. It then sends **Program** a second time, which the daughter stores as a byte array in memory. The daughter is now also in a position to load other transputers, and so on, until the entire network is loaded.

To achieve this, the exploratory worm program is made up of two parts:

```
...   EXE Host     - This runs on the host transputer
...   PROGRAM Worm - This explores the network
```

The Host **EXE** reads the **CODE PROGRAM Worm** fold, and stores it in a byte array **Program**. After resetting the network, it then loads this program onto the first transputer in the network by outputting **Program** on an appropriate link. As the worm proceeds to explore the network, the program running on the host transputer processes any data returned to it from the worm, interpreting and displaying the results.

The following section (section 8.3) describes the **EXE** program which runs on the host transputer, while section 8.4 describes the **PROGRAM** which actually explores the network. Section 8.5 shows some typical results. Section 8.6 provides some notes on extending the exploratory worm for different uses.

In describing the program, declarations and channel protocols have been left out, for brevity, except where they may not be obvious. Variable names start with a lower case letter, constants with a capital. Tokens, indicated by the suffix **.t**, are used to communicate a particular meaning on a channel, for example, **NoMoreData.t**. Similarly, a suffix **.v** is used to indicate a particular interpretation of a stored value, for example, assigning the value **UnAttached.v** to a word which describes the status of a link.

It is assumed that each transputer can access enough memory to run the exploratory worm — information about the memory requirements may be obtained by creating a configuration information fold for the **PROGRAM**.

8.3   The host transputer `EXE`

The program which runs on the host transputer looks like this:

```
SEQ
  code.fold.reader (Screen, from.user.filer[0], to.user.filer[0],
                    programTable, programLength, errorFlag)
  IF
    errorFlag
      SKIP
    TRUE
      SEQ
        ...  Determine which link to examine
        ...  Reset subsystem, links

        -- Main section
        VAL Program IS [programTable FROM 0 FOR programLength] :
        PAR
          WormHandler (LinkIn[linkNumber], LinkOut[linkNumber],
                       ToInterface, linkNumber, Delay, Program)
          Interface (ToInterface, SoftScreen, Heading, linkNumber)
          ...  Display and file output using standard procs

        write.full.string (Screen, "*C*NType <any> to continue")
        Keyboard ? word
```

After determining which of the host transputer's links is to be explored, and resetting the subsystem network, the main section of the program is structured as in figure 8.2. The components are described in the following sections.
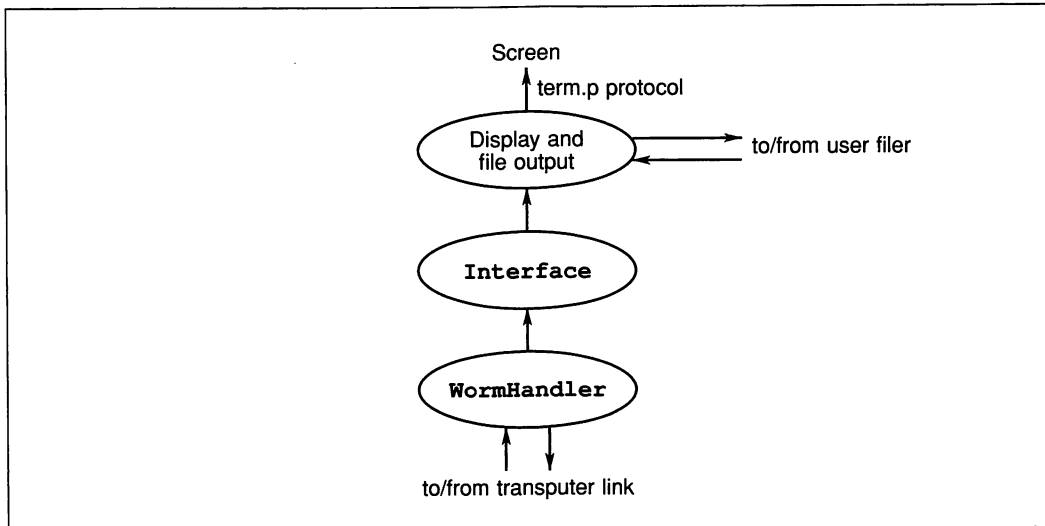
Figure 8.2

8.3.1   Reading the `CODE PROGRAM` fold

The process `code.fold.reader` provided in the example exploratory worm program will attempt to read a `CODE PROGRAM` fold from inside a fold bundle, which may be a compiled or uncompiled **PROGRAM** fold, or a plain text fold. The latter option is included for reasons which are described in the section on filing the output.

The reading and writing of folds and files is described in [1]. If an error occurs, the boolean **errorFlag** is set to **TRUE**, and the cause of the error is displayed on channel Screen, using the term.p protocol.

8.3.2   Resetting the subsystem

It is assumed that the reset pins of the subsystem network are chained together, and controlled by the host transputer (for example, the Subsystem Reset pin on a B004, as described in [2]). In order to reset the transputers correctly, the reset pin must be held high for a certain minimum period of time — a millisecond is ample.

8.3.3   Determine which link to examine

The program asks the user which link of the host transputer, **linkNumber**, is to be examined — the link which is connected to the subsystem must be stated. None of the other links will be tried during the course of the program. If two (or more) links are connected to the same subsystem, then only one can be tried. In this case, the other link will receive data from the subsystem, as the worm program explores, which remains unacknowledged. In order that this does not upset any program running on the host transputer after the exploratory worm has completed, all the links are reset on completion of the program. The resetting of links is described in [3].

8.3.4   Worm handler

The channels **LinkIn, LinkOut** have been placed at the transputer's hard links. This process attempts to load a transputer connected to link **linkNumber** with the exploratory worm program. However, there may be nothing connected at all, or the transputer connected may not have been reset, or not powered on, or some other simple problem, in which case the output will fail. To cater for this eventuality, the **OutputOrFail** routines described in [3] are used. If the output of the code **Program** is not completed within a period **Delay**, then it is abandoned, and the link is reset. This makes it possible for the program to terminate neatly, even if there is no transputer connected to the link.

If the code **Program** is successfully output from the link, booting a transputer, then **PROC WormHandler** sends more data, as described in section 8.4.3. In particular, this new transputer is given an identity number '0'. As the exploration proceeds, **PROC WormHandler** relays data back from the network to **PROC Interface**.

8.3.5   Interface

The Interface process is passed data from the worm handler. This is interpreted, and text is output on channel **SoftScreen** using the term.p protocol [1].

8.3.6   Display and file output

The output from **PROC Interface** is suitable for immediate display on the screen. However, the standard library processes **scrstream.fan.out** and **scrstream.to.file** are used to file a copy of the output. To do this, the user must transfer the **CODE PROGRAM** fold from the **PROGRAM Worm** fold into an empty text fold. When the **EXE** is run, pointing at this text fold, then a new, filed fold will be created which contains the output from **PROC Interface**:

```
{{{  Results
...F CODE PROGRAM Worm
...F Output will appear here
}}}
```

write.endstream is used to close down these processes.

If the program is run while pointing at a **PROGRAM** fold, results are displayed but not filed.

8.4   The exploratory worm PROGRAM

8.4.1   Introduction

As described in section 8.2, the exploratory worm program is constructed as a **PROGRAM** fold which consists of a separately compiled process, **SC Worm**, placed on a single transputer. This is then extracted to produce a **CODE PROGRAM Worm** fold, which contains code to boot a transputer and run **SC Worm** on that transputer. This section now describes how that **SC** is constructed.

The exploratory worm is structured as follows :

```
SEQ
    ...  Read in copy of program, identify boot link
    ...  Initialise

    SEQ I = 0 FOR NLinks
      ...  Try each link in turn
    ...  Return control to parent

    ... Feed back final link information to parent
```

When **SC Worm** starts to run on a transputer, it first identifies which link is connected to its parent, i.e. which of its neighbours booted it, and inputs a copy of the program code so that it, too, may boot other transputers.

After initialising various flags (which keep track of which links have been explored, etc.), the program now picks a link, and tries to send a probe down the link, which may (or may not) be connected to another transputer. An **OutputOrFail** routine is again used, and if the program does not receive any response, it will timeout and look elsewhere.

The period of time for which program is prepared to wait, **Delay**, is quite critical. It must be long enough for any neighbour to have the chance to reply, but not so long that the program is slow to explore a large network of transputers. A **Delay** of 30 milliseconds has been found to be appropriate.

Section 8.4.2 describes the way in which a transputer probes a link to test whether a neighbouring transputer is attached. Section 8.4.3 describes how, if this is successful, the program is loaded and run on the neighbour. These are incorporated into the exploration worm in section 8.4.4, which describes a simple algorithm for exploring a tree of transputers. In section 8.4.5, this algorithm is generalised, to enable the exploration of a general network of transputers.

8.4.2   Probing a neighbouring transputer

A transputer can conveniently test whether link I is attached to an unbooted neighbouring transputer by using the Peek and Poke feature [4]. For example, it may load a word of data at an address, and then read it back, as follows:

```
[4]CHAN OF ANY LinkIn, LinkOut :
PLACE LinkIn  AT 4 :
PLACE LinkOut AT 0 :
SEQ
   LinkIn[I]   ! 0(BYTE); Address; Data   -- Poke
   LinkIn[I]   ! 1(BYTE); Address         -- Peek
   LinkOut[I]  ? word                     -- Data is returned
```

Provided that the address specified exists in memory, then the word returned should match the data sent. A suitable address is MinInt, the minimum 32-bit integer, i.e. #80000000, the bottom of the neighbouring transputer's internal RAM.

In practice, an **OutputOrFail** routine is used for peeking and poking, in case the link is unattached. If successful, the Data is returned on hard channel **LinkIn[I]**. Otherwise, (after a time **Delay** has elapsed,) the program assumes that the link is unattached.

8.4.3   Booting a neighbouring transputer

Having determined that a link is connected to an unbooted neighbour, a transputer loads a neighbouring, unbooted transputer by outputting the code **Program**, as mentioned in section 8.2. The newly booted neighbour will first read in a copy of the program, and identify the boot link:

```
SEQ
  ALT I = 0 FOR 4   -- Determine which link is connected
                    --          to my parent!
    LinkIn[I] ? programLength
      parentLink := I

  LinkIn[parentLink] ? [programTable FROM 0 FOR programLength]
  LinkIn[parentLink] ? token; loadingData

  loadingData[3] := parentLink
  LinkOut[parentLink] ! LoadingData.t; loadingData

  LinkIn[parentLink] ? token   -- Synchronise.t token from the host
```

The parent sends the length of the program, which enables the daughter to determine which link is connected to the parent. The code **Program** is sent again, and stored by the daughter as a byte array for future use. The parent also sends a set of data which includes the parent identity number, the link attached to the daughter, and the number of transputers found so far, **nTransputers**. The daughter returns the data, with the link on which the daughter was booted appended.

The data returned by the daughter is referred to as **loadingData**. **loadingData** contains information useful to follow the path of the worm. Its four elements are, in order, the identity number of the parent, the link which the parent used to boot the daughter, the identity number of the daughter, and the link on which the daughter was booted. This array is transmitted back to the host transputer for display. The **WormHandler** process, running on the host, acknowledges receipt of the **loadingData** with a **Synchronise.t** token, transmitted back to the new daughter.

8.4.4   Exploring a tree of transputers

This section describes a simplified version of the exploration algorithm, suitable for exploring a tree, i.e. a network in which there are no closed loops. The complete algorithm is described in section 8.4.5. An example of a tree of transputers is shown in figure 8.3.

The worm explores the branches of the tree sequentially. Excluding the host transputer, each transputer in the tree will be in one of the following states:

   (R) reset but unbooted;

   (0) booted, but not yet probing its links;

   (1) probing a link, to see if there is another transputer connected;

   (2) booting a neighbouring transputer;

   (3) relaying **loadingData** to the host;

   (4) all links have been explored.

The network is then explored as follows:

Consider figure 8.3 as an example. Suppose that link 3 of transputer A has booted transputer B by link 0, and B has input a copy of the program from A. A enters stage 3, in which it will wait passively to transmit further data. Transputer B starts stage 1, probing one of its links to see if any other transputer is connected. Since link 0 is known to be connected to transputer A, link 1 is the first link to be probed. As described in section 8.4.1, the nucleus attempts to poke and then peek any transputer which may be attached to that link.
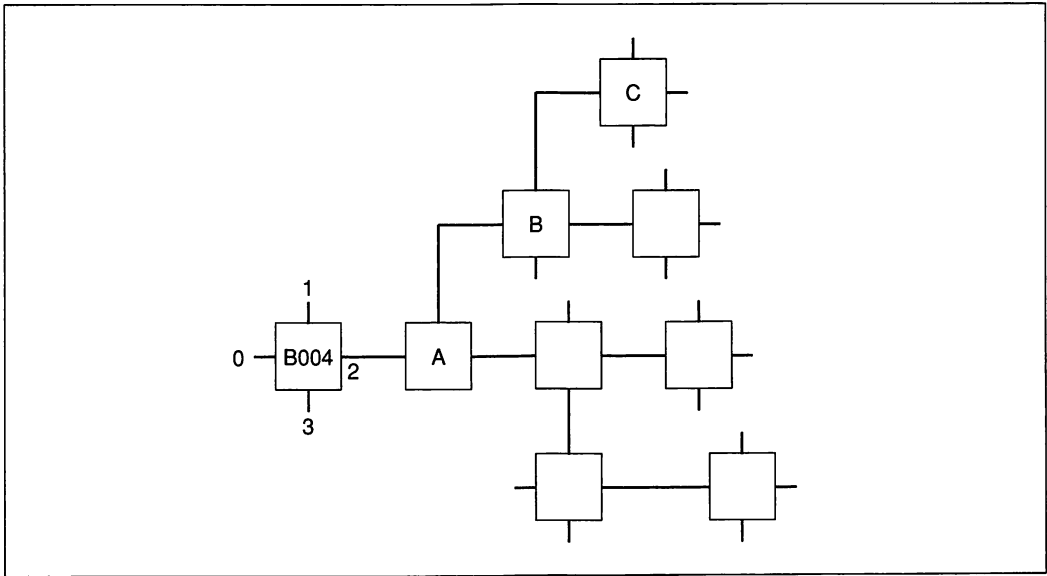
Figure 8.3

The nucleus then waits for a word (which should be MinInt), to be returned on input link 0, for a period of time, **Delay**, before timing out. If nothing is returned, the program assumes this link is unattached, and sets a boolean **downLoad[0]** to **FALSE**. The next link, link 2, is probed in a similar manner.

However, let us assume that a transputer is attached to link 1, and that it has returned the value MinInt in response to the probing. Transputer B now attempts to load the neighbour with code (stage 2), as described in the previous section.

Call this new daughter 'C'. C determines its **parentLink**, the code **Program**, and **loadingData** (stage 0). It takes its identity number to be **nTransputers**, and increments **nTransputers** by one, where **nTransputers** is the number of transputers found so far (the third element of **loadingData**).

At this point, transputer B enters stage 3 of the program, and acts simply to pass on messages from C, even though it has not yet checked links 2 or 3. While transputer C explores its environment, B does not attempt to timeout link 1. Let us suppose that C is not connected to any other transputers. Having failed to find any neighbours, transputer C returns control to B, by sending the token **ReturnControl.t**, together with the latest number of transputers found so far. Transputer C then enters stage 4, and since it has tried all of its links, takes no further part in the exploration. B sets **downLoad[1]** to **TRUE**, to note that a transputer has been loaded from this link.

Transputer B now returns to stage 1 of the program, and similarly tries link 2, and finally link 3. When all links have been tried, B returns control to A, together with the number of transputers found so far. And so on ...

Because of the sequential nature of the algorithm, there is only ever one process actively testing its links. That transputer alone stores the correct value of **nTransputers**. This enables a unique identity number to be given to each transputer as the exploration proceeds.

If a transputer is booted on link `parentLink`, then the above algorithm may be expressed as follows:

```
SEQ
  SEQ I = 0 FOR 4
    downLoad[I] := FALSE
  nTransputers  := LoadingData[2]
  id            := nTransputers
  nTransputers  := nTransputers + 1
  SEQ I = 0 FOR 4                              -- Try each link in turn
    IF
      I = parentLink
        SKIP
      TRUE
        SEQ
          stage   := 1
          waiting := FALSE
          badOut  := FALSE
          ...  Probe neighbouring transputer (set waiting)     (i)
          ...  Boot neighbour, and wait while worm explores   (iii)

  LinkOut[parentLink] ! ReturnControl.t; nTransputers
```

Note:

(i) Peek and poke a neighbour:

```
SEQ
  OutputToken.t (LinkOut[I], 0(BYTE), Delay, badOut)          -- (ii)
  OutputInt.t   (LinkOut[I], MinInt,  Delay, badOut)
  OutputInt.t   (LinkOut[I], MinInt,  Delay, badOut)
  OutputToken.t (LinkOut[I], 1(BYTE), Delay, badOut)
  OutputInt.t   (LinkOut[I], MinInt,  Delay, badOut)

  Clock ? time
  ALT
    LinkIn[I] ? token  -- Value returned
      SEQ
        stage := 2
        waiting := TRUE
    Clock ? AFTER time PLUS Delay
      SKIP
```

Note how the return of the value MinInt indicates that a successful poke and peek has taken place (the boolean `badOut` also indicates that this transputer has output the peek and poke). `waiting` is now set to true, and the algorithm enters the next loop.

(ii) The procs `OutputToken.t`, `OutputInt.t`, `OutputString.t` are based on the output or fail routine. For example:

```
PROC OutputToken.t (CHAN OF ANY ToLink, VAL BYTE Token,
                                VAL INT Delay, BOOL stopping)
  INT time :
  TIMER Clock :
  VAL [1]BYTE String RETYPES Token :
  IF
    stopping
      SKIP
    TRUE
      SEQ
        Clock ? time
        time := time PLUS Delay
        OutputOrFail.t (ToLink, String, Clock, time, stopping)
  :
```

(iii) Given the success of (i) (`waiting` is set to `TRUE`), now try to boot the neighbouring transputer:

```
SEQ
  ...    Try to boot neighbouring transputer
  WHILE waiting        -- worm explores branch off neighbour
    LinkIn[I] ? token
      CASE token
        ...   LoadingData.t                                     (iv)
        ...   ReturnControl.t                                   (v)
```

Booting is performed as follows:

```
VAL []BYTE InitialData RETYPES [Id, I, nTransputers, 0] :
VAL Program IS [programTable FROM 0 FOR programLength] :
SEQ
  OutputString.t (LinkOut[I],         Program,  Delay, badOut)
  OutputInt.t    (LinkOut[I], SIZE Program,  Delay, badOut)
  OutputString.t (LinkOut[I],         Program,  Delay, badOut)
  OutputInt.t    (LinkOut[I], LoadingData.t, Delay, badOut)
  OutputString.t (LinkOut[I], InitialData,   Delay, badOut)
```

Although we know, from peeking and poking, that there is a transputer waiting to be booted off this link, it helps debugging to use the output or fail routines again here!

(iv) The `loadingData` is returned to the host (for immediate display) and is acknowledged by the token `Synchronise.t`. On receipt of the data, the host process returns the token `Synchronise.t`. This synchronisation is important, for it guarantees that all transputers at stage 3 are ready to be probed on any link J, and are not still engaged in returning `loadingData`.

```
LoadingData.t
  [LoadingDataLength]INT passOnData :
  SEQ
    LinkIn[I]             ? passOnData
    LinkOut[parentLink] ! LoadingData.t; passOnData
    LinkIn[parentLink]  ? token                 -- Synchronise.t
    LinkOut[I]          ! Synchronise.t
    stage := 3
```

(v) The return of control indicates that the tree off link **I** has been completely explored. This process may now explore other links.

```
ReturnControl.t
   SEQ
      LinkIn[I] ? nTransputers
      downLoad[I] := TRUE
      waiting := FALSE
```

Error reporting will be described in the next section.

The searching procedure is initiated by **PROC WormHandler** booting the first transputer in the tree, and telling it that **nTransputers** = 0. When that transputer finally returns control to **WormHandler**, the total number of transputers in the network will be returned, and the network will have been completely searched.

8.4.5   Exploring a general network of transputers

The algorithm described in the previous section would be quite satisfactory if all networks took the form of a tree. However, they are usually more complicated, in that they may have either or both (i) two links connected on the same transputer, and (ii) there are closed loops of connections involving more than one transputer. The network will still have a unique start point, however, namely the host transputer. An example is shown in figure 8.4.
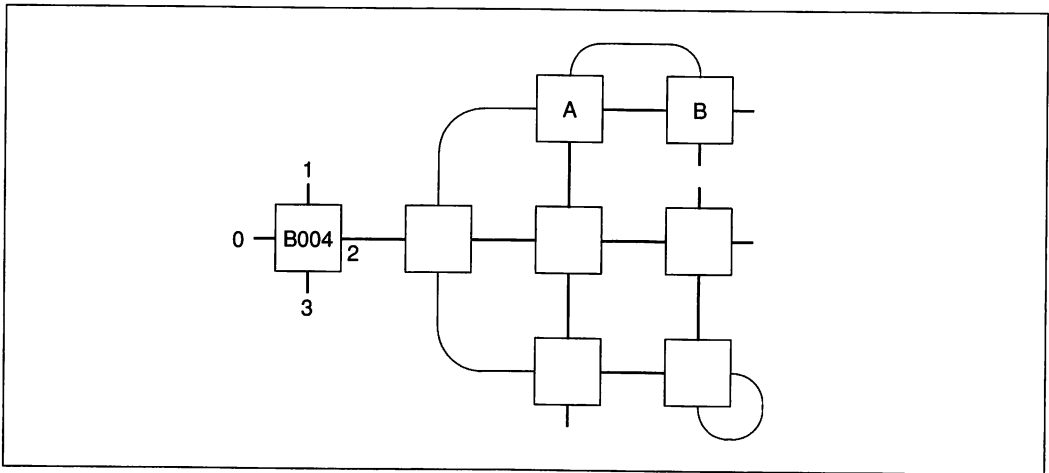
Figure 8.4

The basic algorithm is as before, but in addition there is the situation where a link is connected back to a transputer which has already been booted. This is handled by arranging for every transputer to 'listen' on all links which have not yet been tried — using a replicated **ALT** construct.

Suppose, for example, that link 2 of transputer A has booted transputer B on link 0, and is now passively waiting while B explores further. B outputs the poke and peek sequence on link 1, which arrives back at link 1 of transputer A. It must now be arranged that A will recognise this sequence, even though it comes in on a different link to the one on which daughter B was booted. So A inputs the whole message, and returns a token **AlreadyLoaded.t**, which has a value different from **MinInt**, in order to be recognised by B.

In order that A does not try link 1 again later, a boolean **tryLink[I]** is maintained (initialised to true), indicating whether to try probing off link **I**. In our example, **tryLink[1]** is set to **FALSE**.

It is also useful at this stage to build up a map of which links are connected to whom. A table, **[4][2]INT linkArray,** is assembled for each transputer, in which each link has a corresponding entry giving the

identity of the neighbour attached to that link (if any), and that neighbour's link. For example,

```
linkArray[3] := [6,0]
```

would be set to indicate that link 3 is connected to link 0 of transputer 6. When a parent boots a daughter, this information is communicated in the `loadingData`, and may be entered into the table as appropriate. However, when a transputer probes another one which is already loaded, the programs running on each transputer must exchange identities and link numbers, storing the information in `linkArray`.

The central part of the program now looks like this:

```
SEQ
  ...  Initialise downLoad, id, nTransputers as before
  ...  Initialise tryLink, linkArray                          (i)
  SEQ I = 0 FOR 4
    IF
      NOT tryLink[I]
        SKIP
      TRUE
        ...  Abbreviations as before
        SEQ
          ...  Initialise as before
          ...  Probe neighbour                                (ii)
          ...  Boot neighbour, and wait for reply             (iv)
          tryLink[I] := FALSE
  LinkOut[parentLink] ! ReturnControl.t; nTransputers
```

Note:

(i) Initialise `tryLink[I]` to `TRUE` for all links except the link back to parent. The elements 0 and 1 of the array `loadingData` contain the identity and link of the parent transputer.

```
SEQ I = 0 FOR 4
  tryLink[I] := TRUE
tryLink[parentLink] := FALSE
linkArray[parentLink] := [loadingData FROM 0 FOR 2]
```

(ii) There is now the possibility that two links on the same transputer are connected. Hence, the peek and poke must be done in parallel to listening on all other links:

```
PAR
  ...  Probe neighbouring transputer
  SEQ
    Clock ? time
    ALT
      ALT J = 0 FOR NLinks
        (J <> I) AND tryLink[J] & LinkIn[J] ? probeString
          SEQ
            linkArray[J] := [id, I]
            linkArray[I] := [id, J]
            tryLink[J] := FALSE
      LinkIn[I] ? token
        CASE token
          ...  MinInt as before
          ...  AlreadyLoaded                                  (iii)
          ...  ELSE  -- error                                 (vi)
      ...  Time out as before
```

(iii) If there is a closed loop (other than 2 links connected on the same transputer), we get the situation that one transputer probes another, which replies **AlreadyLoaded.t**. The two ends then exchange pleasantries, viz id and link.

```
PAR
  LinkOut[link] ! [id, link]
  LinkIn[link]  ? linkArray[link]
```

(iv) As before, **waiting** is only set to true if a neighbouring transputer has been found. The case when two links are connected on the same transputer need not now be considered:

```
SEQ
  ...  Try to boot neighbouring transputer as before
  WHILE waiting
    SEQ
      Clock ? time
      ALT
        ALT J = 0 FOR NLinks
          (J <> I) AND tryLink[J] & LinkIn[J] ? probeString
            ...  Reply 'AlreadyLoaded.t'                    (iii)
        LinkIn ? token
          CASE token
            ...  LoadingData.t                                (v)
            ...  ReturnControl.t                      (as before)
            ...  ELSE -- error                               (vi)
    ...  Time Out                                           (vii)
```

(v) In addition to passing the loading data back, we also keep a note of the daughters id, boot link:

```
IF
  stage = 2
    linkArray[I] := [passOnData FROM 2 FOR 2]
  TRUE
    SKIP
```

(vi) Make a note of the fact that a bad communication has taken place on this link by making a record in **linkArray**. Use a special token **TokenError.v** to indicate that an unexpected token has been returned. A classic cause of this is when two transputers are communicating at different link speeds (10 and 20 MHz, for example).

```
SEQ
  waiting := FALSE
  linkArray [I] := [stage, TokenError.v]
```

(vii) A timeout at stage 1 implies that the link is unattached. However, if a timeout occurs at a later stage, assuming **Delay** is long enough to allow for the booting of a daughter, then the neighbour has not been successfully loaded — report this as an error.

```
Clock ? AFTER time PLUS Delay
  SEQ
    linkArray[I] := [stage, TimeOutError.v]
    waiting := FALSE
```

8.4.6   Returning the local link map

Having explored the local connections of each link on a transputer, and returned control to the parent, we wish to relay the information `linkArray` back to the host transputer. This is done as follows:

```
CHAN OF ANY ToParent IS LinkOut[parentLink] :
SEQ
  stage := 4
  ToParent ! NetworkData.t; id; linkArray

  SEQ I = 0 FOR 4
    IF
      NOT downLoad[I]
        SKIP
      downLoad[I]   -- Pass on network info from daughter processes
        SEQ
          reading := TRUE
          WHILE reading
            SEQ
              LinkIn[I] ? token
              CASE token
                ...  NetworkData.t                              (i)
                ...  NoMoreData.t                               (ii)
                ...  ELSE                                       (iii)
  ToParent ! NoMoreData.t
```

Note:

(i) Pass on the identity and link array.

```
NetworkData.t          -- pass on id and info
  INT passOnId :
  [4][2]INT passOnLinkArray :
  SEQ
    LinkIn[I] ? passOnId; passOnLinkArray
    ToParent  ! NetworkData.t; passOnId; passOnLinkArray
```

(ii) There is no more data to transmit from this branch.

```
NoMoreData.t
  reading := FALSE
```

(iii) This is an error. Return a modified `linkArray` report.

```
ELSE
  SEQ
    reading := FALSE
    linkArray[I] := [stage, TokenError.v]
    ToParent ! NetworkData.t; id; linkArray
```

Data from each transputer, giving the id. number and local link connections, will arrive back at **WormHandler** after the entire network has been loaded.

8.5   An example

Below is some typical output from an exploratory worm program when run on the transputer configuration shown in figure 8.5:
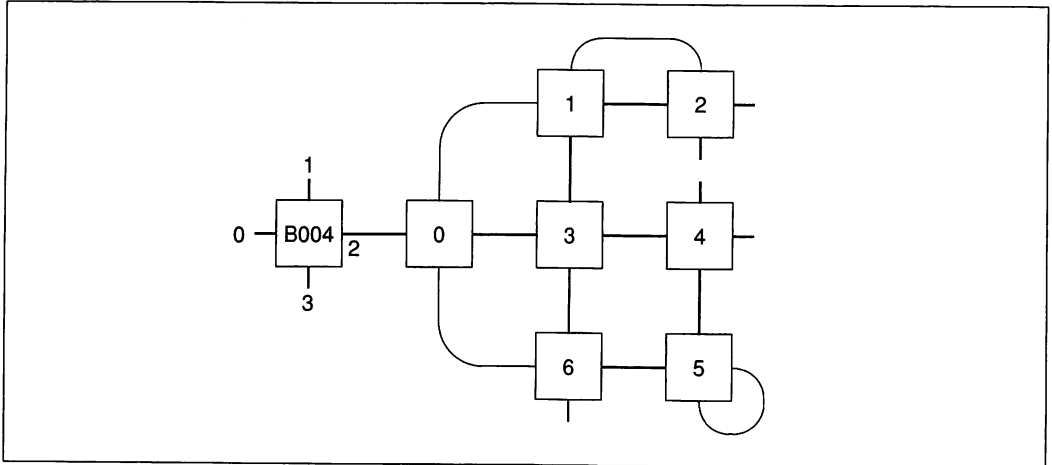
Figure 8.5

```
Checking network off link 2 ...

        Parent      Daughter
      Id   Link     Id   Link
     host    2       0     0
       0     1       1     0
       1     1       2     1
       1     3       3     1
       3     2       4     0
       4     3       5     1
       5     0       6     2

The number of transputers found is 7
Arranged in the following network :

    Id    Link: 0         1         2         3
     0       host-2      1-0       3-0       6-0
     1        0-1        2-1       2-0       3-1
     2        1-2        1-1       ooo       ooo
     3        0-2        1-3       4-0       6-1
     4        3-2        ooo       ooo       5-1
     5        6-2        4-3       5-3       5-2
     6        0-3        3-3   .   5-0       ooo
```

The first table refers to the initial loading of the network. It indicates that link 2 of the host transputer (running on a B004, for example) has booted transputer 0 by link 0. Then link 1 of transputer 0 booted transputer 1 by link 0, and so on.

The second table summarises the connectivity of the network, by stating what each link of each transputer is attached to. For example, the entry 6-0 in row 0, column 3, indicates that link 3 of transputer 0 is attached to link 0 of transputer 6.

8.6   Some points to note

This section will note some further developments which can be made to an exploratory worm program, and restrictions on such a program.

8.6.1   16 and 32-Bit compatible programs

The instruction set of the INMOS transputer is independent of the wordlength of the transputer on which it is to run. Code compiled for the IMS T414 may be run on a T800, or T212, for example, provided the following points are observed:

1   If data, for example text strings or constant definitions, is included in the program, then it will be 'word aligned' in the compiled code. A program containing such data, and compiled for a 32-bit transputer ('T4'), will run on a 16-bit transputer ('T2'), but the converse may not be true. Therefore, it will be assumed that programs intended to run on either a T2 or T4 are compiled using the T4 compiler.

2   Communication between two transputers with different word lengths requires a mutually agreed datalength. For example, it might be arranged that all data is input and output as INT16 words, and that `linkArray` is built up and transmitted as an INT16 array.

Internal communication of words should be treated similarly. For example, the input of a word, when compiled for a 32-bit machine, always attempts to input explicitly 4 bytes — which is not what is wanted if the program is to be run on a 16-bit machine.

Beware that, if INT32 words are specified in a program which is compiled for a 32-bit transputer, they will be recognised as being of the natural wordlength of the machine, and no special treatment will be given. If the same code is run on a 16-bit machine without recompilation, the data would be treated as 16-bits, which would be catastrophic if it was intended to communicate a 32-bit word.

3   Peeking and poking of a transputer assumes knowledge of the wordlength of that device. But when a transputer first explores its links, it knows nothing about what is connected at the other end! The simplest way around this is to attempt to poke and peek a neighbour assuming that it is a T2. If this fails, terminate the T2 sequence with an extra byte to make it look like a T4 poke. Then try again for a T4. For example:

```
ToLink ! #00; #00; #80; #00; #80;
         #01; #00; #80                           --   (i)
ToLink ! #00                                     --  (ii)
ToLink ! #00; #00; #00; #00; #80; #00; #00; #00; #80;
         #01; #00; #00; #00; #80                 -- (iii)
```

(i) is a sequence for poking and peeking a 16-bit transputer, (ii) rounds this off to a valid 32-bit poke (but at an address in external memory, which is not guaranteed to exist) and (iii) is a sequence for poking and peeking a 32-bit transputer. Words have been expressed as bytes, little end first, to prevent any possible confusion over compiling 16 and 32-bit words. If the neighbour is already loaded, it should be made to reply immediately it receives probe (i).

4   The memory requirement of programs is determined by the compiler as the number of words needed. However, running a program on a 16-bit transputer may require more words of storage than if the same program was run on a 32-bit transputer. For example, `[4]BYTE array` requires 1 word of storage on a T4, but 2 words on a T2. Since, as is noted in (1) above, the program must be compiled for a 32-bit transputer, the allocation of storage must be forced to be suitable for 16-bit transputers by declaring arrays as follows:

```
[2][ArraySize]BYTE dummyArray :
[ArraySize]BYTE array IS dummyArray[0] :
```

The same applies for boolean and INT16 arrays.

5   Provided that it does not contain any floating point or extended arithmetic, a program compiled and extracted for a T414 will run on a T800. The reverse is not true — do not try to run a program compiled for the T800 on a T414.

6 The code which loads a **CODE PROGRAM** fold onto a transputer is wordlength independent, and a program compiled and extracted to load a T4 will work equally well on a T2, provided that the above points have been noted.

7 Because of differences in code placement, the debugger won't work when the worm is running on a transputer other than the one it was compiled for.

8.6.2   Using an exploratory worm program to perform testing

An exploratory worm program is an extremely useful vehicle for testing transputer based products. Tests for memory and the links may be included in the basic program, for example. If a hardware fault occurs, the program may report the location and nature of the problem, while continuing to test other components in the network. This is particularly useful during a long burn-in run. By testing the network repeatedly with an exploratory worm, any failure may be detected and logged, while the rest of the network continues to be burnt-in.

All INMOS transputers and transputer evaluation boards are burnt-in before shipping, and subsequent failure is unlikely. However, this technique may be useful for testing products which use transputers as components. In designing an exploratory test program, the following points should be borne in mind:

1 The same program will be loaded onto every transputer. Ideally, all components of the network to be tested will be identical, but if there is any variation, the program will have to dynamically assess the attributes (for example memory size, peripherals) of each transputer it finds.

2 The program has its own algorithm for assigning identity numbers to each transputer in the network, which may be quite different to the one which the user has in mind. If a failure occurs, and the program is run again, yet another different numbering of the network may occur.

3 If memory is to be tested, a transputer should test a section of memory of a potential daughter using peek and poke, before booting that daughter. The section tested is the area where the program and workspace will go.

4 If the links are to be tested, it should be remembered that corruption of data on a link (by noise, for example) might cause a data packet to look like an acknowledge, or vice-versa. The **OutputOrFail** predefines are useful in this context.

8.6.3   Using an exploratory worm program to load another program

Another field in which it is useful to have a vehicle to load an arbitrary network is when the user intends to run a program replicated over an array of processors, but does not care too much about their precise configuration. An example of this is the data farm approach to processing [5]. In this, one central processor 'farms out' work to an array of 'worker' processes, each of which is capable of processing a piece of data and returning it. The following points should be made:

1 The program which the user wishes to run on every transputer is included as part of the **SC Worm**, so that it executes after the exploration phase has been completed.

2 An identical program will run on each transputer in the network. This program will be passed information by the exploratory worm such as which links are connected to neighbours, and which is connected back to the parent. From such information, algorithms to control the broadcasting or routing of data may be developed.

3 The host transputer will be responsible for communicating with the rest of the network as required, for example by sending out data for processing, and receiving results back.

4 Although this technnical note has described an exploratory worm as being initiated from the host transputer, there is no reason why it could not be launched out from an already partially loaded system.

A more flexible system can be constructed by arranging that the worm declares a large workspace. After the system has been explored, the host sends out processes, in the form of pieces of compiled code, to specified processors in the network, which are run using **KERNEL.RUN**. This allows the placement of code to be decided at run-time, which might be useful, for example, in constructing a program which takes advantage of all the processors in an arbitrary network, or to be used as a basis for a multi-tasking operating system.

8.6.4   Debugging an exploratory worm program

By its very nature, a worm program is difficult to debug. While the INMOS software debugger is very useful for debugging a program which has been configured to match a known multiprocessor configuration, it does not deal with a program which has explored an unknown network. To make things simpler, let us assume that the program to be debugged is being run on a network of transputers whose configuration is actually known, and which is known to be free from hardware bugs.

Since the worm takes the form of a **PROGRAM** configured for one transputer, a bug which occurs on the first transputer in the network can be traced by using the debugger in the normal way — simply point it at the worm **PROGRAM** and it will give the values of all variables, channel communication, etc., and the point at which the program failed.

If a bug occurs deeper down in the network, use the following procedure. First modify the program so that it looks like this:

```
...  SC Worm
CHAN OF ANY a,b,c,d,e,f,g,h :
PROCESSOR 0 T4
   ...  PLACE a AT 0, b AT 1, etc.
   Worm (a,b,c,d,e,f,g,h)
```

(The channels a, ... h are not used by the worm, but must be declared to ensure that code is placed in the same way as below.)

Now take a copy of this program, and configure it to match the actual network (or part of the network). For example, for a 2 transputer network connected by link 0 on each transputer:

```
...  SC Worm
CHAN OF ANY a,b,c,d,e,f,g,h :
CHAN OF ANY i,j,k,l,m,n :
PLACED PAR
   PROCESSOR 0 T4
      ...  PLACE a AT 0, b AT 1, etc. as before
      Worm (a,b,c,d,e,f,g,h)
   PROCESSOR 1 T4
      ...  PLACE e AT 0, a AT 4, i AT 1, etc.
      Worm (e,i,j,k,a,l,m,n)
```

Load the network by pointing the **EXE** at the Worm **PROGRAM** configured for one transputer, in the usual way. (A suspected software bug occurs which causes the program to fail...) Now point the debugger at the copy of the program configured to match the network. The debugger will give complete symbollic information about the state of the system when the program crashed.

Remember that, even if the failure is severe enough to cause the host transputer to lock up, so that it has to be rebooted, the state of the subsystem is not altered by rebooting, and it can still be debugged as above

It is always important that channels are declared and placed on hard links in the same way, no matter how the program is configured. This is to ensure that the way the code is loaded exactly matches the placement of the code for the configured program, as used by the debugger. If in doubt, use the 'check code' feature of the debugger to check that placement of the code loaded on the transputer matches the configured program.

8.6.5   Loading a network in parallel

Section 8.4 described an algorithm for sequentially exploring a network. This is quite fast enough for most purposes. However, if a large program is to be loaded onto an extremely large network of transputers, a parallel loading algorithm might be considered. Such an algorithm is not so simple as the one described above. In particular, it may happen that two loaded transputers simultaneously try to boot a third, unloaded transputer, which is connected to both of them. The following points should be noted:

1 After receiving a peek or poke sequence on a particular link, an unbooted transputer will continue to listen on all links for any further communication. Therefore, if two different transputers probe the same daughter, confusion may arise. In particular, it would be impossible to test the memory properly by peeking and poking.

2 Once a transputer has been successfully booted, care must be taken in how it identifies its parent. For another transputer, besides the genuine parent, may also be trying to boot the new daughter.

3 The numbering of each transputer with unique identity numbers can only take place after the entire network has been explored.

8.7   References

1 *Transputer Development System*, Prentice Hall 1988.

2 *IMS B004 Evaluation Board User Manual*, INMOS Ltd

3 *Extraordinary use of transputer links*, Technical note 1, INMOS Ltd 1987.

4 *Transputer Reference Manual*, Prentice Hall 1988.

5 *Communicating Process Computers*, Technical note 22, INMOS Ltd, 1987.

9 Extraordinary use of transputer links

9.1 Introduction

The transputer link architecture provides ease of use and compatibility across the range of transputer products. The transputer link is asynchronous at the bit level, which removes the need to distribute a clock within tight phase constraints; indeed, separate clocks can be used to supply the transputers within a system. The use of a handshaken protocol at the byte level allows fast systems to communicate with slow systems without overrun problems. Finally, the provision of synchronised communcation at the message level matches the occam model of communication.

Transputer links are intended to be used for communication within a system of devices connected on the same PCB or via a backplane. The links are TTL compatible. This allows the use of simple buffers and determines their DC noise margins. If transputer links are used within their specifications (Vcc, clock jitter, clock frequency, data skew, and decoupling) they are extremely reliable; there will no run out errors on clocking and the synchronisation failure rate has been designed to be less than 1 failure per 10**25 samples.

In certain circumstances, such as communication between a development system and a target system, or for communication via an unreliable interconnect, it is desirable to use a transputer link even though the synchronised message passing of occam is not exactly what is required. Such extraordinary use of transputer links is possible but requires careful programming and the use of some special pre-defined occam procedures. This note explains how to use these procedures and gives two examples of their use.

9.2 Clarification of requirements

It is essential to have a clear idea of the requirements of a system in order to program extraordinary use of the transputer links. We have two cases to consider here. The first is of a system consisting of two distinct parts connected via a link. Here the requirement is to insulate each system from the other, perhaps allowing one system to monitor to behaviour of the other. The second case is of a system which uses an unreliable interconnect, where there is a danger of disconnection, or if the link is used outside its specified noise margins, a danger of data corruption.

9.2.1 Connection of distinct sub-systems

As an example, consider a development system connected via a link to a target system. The development system compiles and loads programs onto the target and also provides the program executing in the target with access to facilities such as a file store. Suppose the target halts (due to a bug) whilst it is engaged in communication with the development system. The development system then has to analyse the target system.

A problem will arise if the development system is written in 'pure' occam. It is possible that when the target system halts, the development system is in the middle of communicating. As a result, the input or output process will not terminate and the development system will be unable to continue. This problem can occur even where an input occurs in an alternative construct together with a timeout (as illustrated below). When the first byte of a message is received the process performing the alternative commits to inputting; the timer guard cannot subsequently be selected. Hence, if insufficient data is transmitted the input will not terminate.

```
ALT
  TIME ? AFTER timeout
    . . .
  from.other.system ? message
    . . .
```

It is important to note that the problem arises from the need to *recover* from the communication failure. It is perfectly straightforward to *detect* the failure within 'pure' occam, and this is quite sufficient for implementing resilient systems with multiple redundancy.

9.2.2   Communication via an unreliable interconnect

In the case of communication via an unreliable interconnect there are a number of possible failure modes. If the interconnect becomes disconnected whilst a data transfer is in progress the communication will not complete. It is possible that this might manifest itself to only one of the systems; if the disconnection occurs after all the data packets have been transmitted but before the final acknowledge packet has been transmitted then the inputting system will see a completed transfer but the outputting system will hang. It is also possible for a disconnection to cause data corruption or the conversion of a data packet into an acknowledge packet (see next paragraph).

If a link is being used outside its noise margins there are a number of errors which may occur. The first is the corruption of the content of a data packet which will lead to the reception of erroneous data. This may be detected by the use of standard checking techniques such as checksums or CRCs. Otherwise, an error will involve the generation of, the deletion of, or the corruption of a packet. This will lead to the breakdown of the end-to-end synchronisation of the protocol, and ultimately, will cause one, or both, of the communicating processes to hang on a communication.

For example, if a data packet is lost, it will not be acknowledged by the receiving transputer. Hence, the transmitting transputer will neither be able to transmit any further data packets, nor to schedule the outputting process. Consequently, the receiving transputer will never receive suffient data packets to schedule the inputting process. Hence neither the inputting process, nor the outputting process will terminate.

9.3   Programming concerns

The first concern of a designer is to understand how to recognise the occurence of a failure. This will depend on the system; for example, in some cases a timeout may be appropriate.

The second concern is to use ensure that even if a communication fails, all input processes and output processes will terminate. As this cannot be achieved directly in occam, INMOS provides a number of predefined procedures which perform the required function. These are described below.

The final concern is to be able to recover from the failure and to re-establish communication on the link. This involves reinitialising the link hardware; again INMOS provides a suitable pre-defined procedure to allow this to be performed.

9.4   Predefined input and output procedures

There are four predefined procedures which implement input and output processes which can be made to terminate even when there is a communication failure. They will terminate either as the result of the communcation completing, or as the result of the failure of the communcation being recognised. Two procedures provide input and output where communication failure can be detected by a simple timeout, the other two procedures provide input and output where the failure of the communication is signalled to the procedure via a channel. The procedures have a boolean variable as a parameter which is set true if the procedure terminated as a result of communication failure being detected, and is set false otherwise. If the procedure does terminate as a result of communication failure having been detected then the link channel will be reset (see later).

All four predefined procedures take as parameters a link channel **c** (on which the communication is to take place), a byte vector **mess** (which is the object of the communication) and the boolean variable **aborted**. The choice of a byte vector as the parameter to these procedures allows an object of any type to be passed along the channel provided it is retyped first.

The two procedures for communication where failure is detected by a timeout take a timer parameter **TIME**, and an absolute time **t**. The procedures treat the communcation as having failed when the time as measured by the timer **TIME** is **AFTER** the specified time **t**. The names and the parameters of the procedures are:

```
InputOrFail.t(CHAN c, []BYTE mess, TIMER TIME, INT t,
                                        BOOL aborted)
```

and

```
OutputOrFail.t(CHAN c, VAL []BYTE mess, TIMER TIME, INT t,
                                        BOOL aborted)
```

The other two procedures provide communication where failure cannot be detected by a simple timeout. In this case failure must be signalled to the inputting or outputting procedure via a message on the channel **kill**. The message is of type **INT**. The names and parameters to the procedures are:

```
InputOrFail.c(CHAN c, []BYTE mess, CHAN kill, BOOL aborted)
```

and

```
OutputOrFail.c(CHAN c, VAL []BYTE mess, CHAN kill, BOOL aborted)
```

9.5   Recovery from failure

To reuse a link after a communication failure has occurred it is necessary to reinitialise the link hardware. This involves reinitialising both ends of both channels implemented by the link. Furthermore, the reinitialisation must be done after all processes have stopped trying to communicate on the link. So, although the **InputOrFail** and **OutputOrFail** procedures do, themselves, reset the link channel when they abort a transfer, it is necessary to use the fifth pre-defined procedure **Reinitialise(CHAN c)**, after it is known that all activity on the link has ceased.

The **Reinitialise** pre-defined must only be used to reinitialise a link channel after communication has finished. If the procedure is applied to a link channel which is being used for communication the transputer's error flag will be set and subsequent behaviour is undefined.

9.6   Examples: two systems with extraordinary link usage

The following examples illustrate two systems which make extraordinary use of transputer links. The first example is a development system, the second example is of two systems interconnected by a link which may be physically disconnected and re-connected at any time.

9.6.1   Example 1: a development system

The problem

For our example we return to the development system described above.
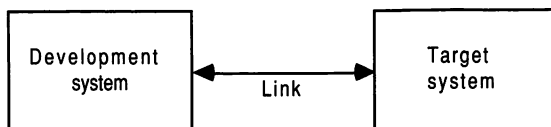
The solution

The first step in the solution is to recognise that the development system knows when a failure might occur, and hence the development system knows when it might be necessary to abort a communication.

We will assume that the process which interfaces to the target system is sent a message when the development system decides to reset the target causing the interface process to abort any transfers in progress. The development system can then reset the target system (which resets the target end of the link) and re-initialise the link.

We can now outline the construction of such a system. The program below would be that part of the development system which runs once the target system starts executing, until such time as the target is reset and the link is reinitialised.

```
SEQ
  CHAN terminate.input, terminate.output :
  PAR
    ...   interface process
    ...   monitor process
  ... reset target system
  Reinitialise(link.in)
  Reinitialise(link.out)
```

The monitor process will output on both **terminate.input** and **terminate.output** when it detects an error in the target system.

The interface process consists of two processes running in parallel, one which outputs to the link, the other which inputs from the link. As the structure of the processes is similar we only discuss the process which outputs to the link. If there were no need to consider the possibility of communication failure the process might be

```
WHILE active
  SEQ
    ...
    ALT
      terminate.out ? any
        active := FALSE
      from.dev.system ? message
        link.out ! message
    ...
```

This process will loop, forwarding input from **from.dev.system** to **link.out**, until it receives a message on **terminate.out**. However, if after this process has attempted to forward a message, the target system halts without inputting, the interface process will fail to terminate.

The following program overcomes this problem:

```
WHILE active
  BOOL aborted :
  SEQ
    ...
    ALT
      terminate.out ? any
        active := FALSE
      from.dev.system ? word
        SEQ
          OutputOrFail.c(link.out, message, terminate.out, aborted)
          active := NOT aborted
```

This program is always prepared to input from `terminate.out`, and is always terminated by an input from `terminate.out`. There are two cases which can occur. The first is that the message is received by the input which then sets `active` to false. The second is that the output gets aborted. In this case the whole process is terminated because the variable `aborted` would then be true.

9.6.2   Example 2: two systems connected by a link

The problem

In this example we consider two transputer-based systems, connected by a link. The particular problem with which we are concerned is that the link between the two systems might become disconnected. (We assume that the electrical design of the system is adequate).

This example illustrates two things. Firstly how to detect that the link has become disconnected, and secondly how to restart communication after it is re-connected.

The solution

The key to this solution is detecting the disconnection of the link. Unlike the development system example we do not straightforwardly know when this has occured. For example, if one system has not received communication from the other system for thirty minutes it cannot necessarily deduce that the link has been disconnected; it may just be that the other system has not tried to communicate for thirty minutes!

To overcome this problem we adopt the use of 'watchdog' processes on each system to ensure that it communicates frequently with the other system. The frequency of communication is chosen so that the disconnection of the link is detected as quickly as is required by a system.

In this solution each system contains a process which interfaces to the communication link. This process connects to an input channel, an output channel and both the channels implemented by the link. The outline of this process is as follows:

```
TIMER TIME :

PROC copier(CHAN output, input, unreliable.in, unreliable.out)
  INT start.time :
  SEQ
    ...   synchronise with other end
    TIME ? start.time
    WHILE active
      SEQ
        ...   copy until failure occurs
        ...   resynchronise
```

For simplicity we will assume that the system starts with the link connected. First, the two systems synchronise by passing a message. This establishes a common timeframe for the two systems (used when we need to re-establish communication after disconnection of the link). Then the systems copy information between themselves until the link is disconnected. If one system detects a failure it ensures that the other system detects a failure by deliberately not engaging in communication for a suitable period. The two systems then attempt to re-establish communication.

The copier performs the copying using two processes running in parallel, as follows:

```
CHAN in.to.out, out.to.in :
PAR
  copy.in (unreliable.in,  output, out.to.in, in.to.out, one.sec)
  copy.out(unreliable.out, input,  in.to.out, out.to.in, one.sec/4)
```
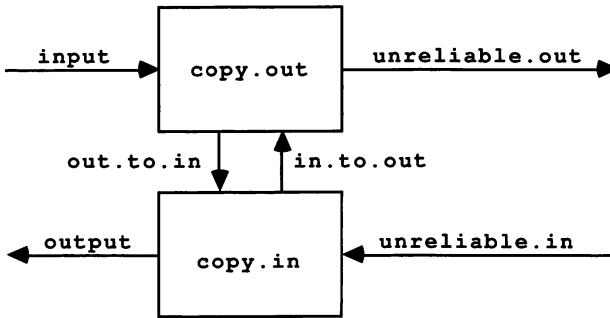

The channels `in.to.out` and `out.to.in` enable each process to signal the other when one detects failure. The processes implement a protocol on the link channels with two types of packet, 'data' and 'tick' packets. A data packet is a 'data' tag, followed by a message, a tick packet consists of just a 'tick' tag. In this example both the tag and the message are one word long.

The processes forward and receive messages as needed and insert tick packets if there are no messages being forwarded. The disconnection of the link is detected either by the input process or the output process failing to communicate within their alloted time.

In this example the outputting process outputs at least once every quarter second (on `unreliable.out`) and assumes that the link has been disconnected if the output does not complete within a quarter second. The inputting process will assume the link has become disconnected if it does not receive a message (on `unreliable.in`) for one second.

The coding of the two procedures `copy.in` and `copy.out` can now be explained. The program text is given in section 9.7. Both procedures (A) declare an integer **mess** and then retype it to a byte array **mess.a**. This allows the integer **mess** to be passed to the predefined procedures which require a byte array as a parameter. The main loop of both procedures (B) continue until either the procedure receives a message which tells it that the other procedure, running in parallel, has detected link disconnection (C), or it has detected an error itself (G).

The other possibilities for the main loop of `copy.out` are to receive a message on channel `output` (E), or to determine that it is time to send a 'tick' (D). In both cases an `OutputOrFail.t` is used in case the link is disconnected whilst `copy.out` is outputting.

If `copy.in` does not receive a message on `error.det` it will perform an input (F). This is done using `InputOrFail.t` which will detect link disconnection if the timeout is exceeded.

Each process contains program to inform the other, parallel, process when it detects an error (G). This runs an input in parallel with an output to ensure that if the other parallel process has performed an output, the communication will occur correctly. Correspondingly, if the procedure is informed that an error has occured by the other process (C) it acknowledges the receipt of that information.

It now remains to describe how to restart communication. The first problem here is to identify that the link has been reconnected. In this example we will assume that there is no way of doing this other than by trying to use the link. (This is not ideal but is adequate).

The scheme we use is for both systems to try, repeatedly, to communicate with the other. We use the transputer's timer to ensure that the systems attempt to communicate at the same time. The systems execute processes of the form

```
WHILE trying
  SEQ
    ...   wait until start of next cycle
    ...   reset both link channels
    ...   wait until next phase of cycle
    PAR
      ... input from link channel with timeout
      ... output to link channel with timeout
    trying := input.failed OR output.failed
```

The breaking of the cycle into distinct, non-overlapping, phases ensures that the systems will not fail to communicate because one system is resetting its links at the same time as the other system is trying to communicate.

The full code is given in section 9.8. In this code `interval` contains the number of timer ticks in a cycle, and `phase` contains the number of ticks in a phase (which equals `interval`/3). The program fragment starting at (A) calculates the time to the start of the next cycle. `delta.time` contains the the elapsed time since the processes originally synchronised (modulo the wordlength). The `LONGDIV` computes the time since the start of the last cycle. Note that in order for this code to work correctly the number of ticks in a cycle must divide 2**wordlength exactly.

9.7   Program listing 1

```
VAL INT data.tag IS 0 :
VAL INT tick.tag IS 1 :

PROC get.next.tick(INT next.tick, VAL INT delta)
  SEQ
    TIME ? next.tick
    next.tick := next.tick PLUS delta
:

PROC copy.out(CHAN out.dubious, input, error.det, error.gen,
              VAL INT delta)
  INT mess :                                             (A)
  []BYTE mess.a RETYPES mess :
  INT next.tick :
  BOOL active :
  SEQ
    active := TRUE
    WHILE active                                         (B)
      INT sink, data :
      BOOL error :
      SEQ
        get.next.tick(next.tick, delta)
        PRI ALT
          error.det ? sink                               (C)
            SEQ
              error.gen ! 0
              active := FALSE
          TIME ? AFTER next.tick                         (D)
            SEQ
              get.next.tick(next.tick, delta)
              mess := tick.tag
              OutputOrFail.t(out.dubious, mess.a,
                              TIME, next.tick, error)
          in ? data                                      (E)
            SEQ
              next.tick := next.tick PLUS delta
              mess := data.tag
              OutputOrFail.t(out.dubious, mess.a,
                              TIME, next.tick, error)
              IF
                error
                  SKIP
                NOT error
                  SEQ
                    get.next.tick(next.tick, delta)
                    mess := data
                    OutputOrFail.t(out.dubious, mess.a,
                                    TIME, next.tick, error)
        IF
          error
            SEQ
              PAR
                error.gen ! 0
                error.det ? data
              active := FALSE                            (G)
          TRUE
            SKIP
:
```

```
PROC copy.in(CHAN in.dubious, output, error.det, error.gen,
             VAL INT delta)
  INT mess :
  []BYTE mess.a RETYPES mess :                          (A)
  INT next.tick :
  BOOL active :
  SEQ
    active := TRUE
    WHILE active                                        (B)
      INT sink :
      BOOL error :
      SEQ
        get.next.tick(next.tick, delta)
        PRI ALT
          error.det ? sink                              (C)
            SEQ
              error.gen ! 0
              active := FALSE
          TRUE & SKIP
            SEQ
              InputOrFail.t(in.dubious, mess.a,         (F)
                               TIME, next.tick, error)
              IF
                error
                  SKIP
                mess = tick.tag
                  SKIP
                mess = data.tag
                  SEQ
                    get.next.tick(next.tick, delta)
                    InputOrFail.t(in.dubious, mess.a,
                                     TIME, next.tick, error)
                    IF -- forward data unless error detected
                      error
                        SKIP
                      TRUE
                        output ! mess
              IF
                error                                   (G)
                  SEQ
                    PAR
                      error.gen ! 0
                      error.det ? sink
                    active := FALSE
                TRUE
                  SKIP
:
```

9.8   Program listing 2

```
INT start.time :
SEQ
  ...  pass initial message and set up start.time
  WHILE active
    SEQ
      ... copy until failure occurs

      [1]BYTE i.byte, o.byte :
      INT time, delta.time, next.cycle, next.phase, cycles :
      BOOL trying :
      SEQ
        -- determine start of next cycle
        TIME ? time                                              (A)
        delta.time := time MINUS start.time
        LONGDIV(cycles, delta.time, 0, delta.time, interval)
        next.cycle := (time MINUS delta.time) PLUS interval

        trying := TRUE
        WHILE trying
          BOOL input.failed, output.failed :
          SEQ
            TIME ? AFTER next.cycle
            ResetChannel(unreliable.in)
            ResetChannel(unreliable.out)

            next.phase := next.cycle PLUS phase
            TIME ? AFTER next.phase

            next.phase := next.phase PLUS phase
            PAR
              InputOrFail.t(unreliable.in, i.byte, TIME,
                                next.phase, input.failed)
              OutputOrFail.t(unreliable.out, o.byte, TIME,
                                next.phase, output.failed)
            trying := input.failed OR output.failed

            next.cycle := nextcycle PLUS interval
```

Applications

10 A transputer based radio-navigation system

10.1 Introduction

The speed and multi-processing capabilities of the transputer make it ideal for demanding signal processing, calculating and control tasks (figure 10.1). A navigation system needs all these facilities, and the LORAN C system, operating at 100kHz, gives the opportunity for the transputer to capture the incoming radio frequency in real time, without demodulation.
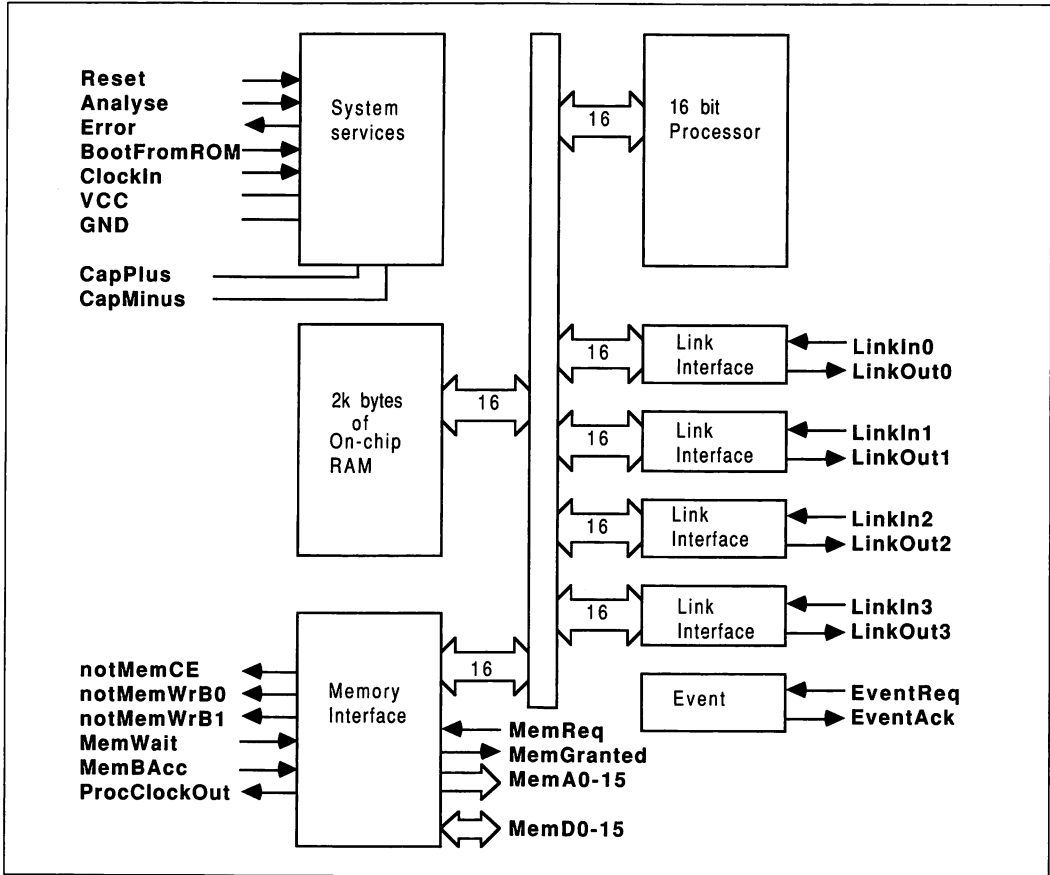
Figure 10.1 The T212 16 bit transputer

The T212 transputer is the 16-bit member of the transputer family. It has 2K bytes of 50ns static RAM on the chip, which allow it to operate at over 10 MIPS. The memory can be extended externally, the external interface being optimised for static memory, with separate address and data lines. Thus Static RAM or program ROM can be attached with no TTL glue logic. The T212 has four serial links operating at 10 or 20 Mbaud rates, designed for connections between transputers or to peripherals such as link adapters. These links have full duplex DMA into or out of the transputer memory, giving the processor the equivalent of eight high-speed DMA controllers on chip. Also on the transputer are a hardware scheduler and timer, and all these taken with the language occam make it a very powerful general purpose processor.

10.2   LORAN

The incentive to design a LORAN C system is given by the imminent opening of a new experimental chain of transmitters serving Northern Europe. The system already covers most of the worlds ocean's, and also the Mediterranean, but southern Britain has lacked coverage.

LORAN (LOng-range RAdio Navigation) is a system run for ships and aircraft by the US government. Like the Decca system, it works by measuring the relative delays from several transmitters, but being long-range, it has far fewer chains, operating at much lower frequency, and no charge is made for its use.

All the transmitters operate on one frequency, but they transmit at a low duty cycle with each chain having a different repetition rate. Thus the receiver can identify the valid signals as those operating at the desired rate, and although one particular signal may be blotted out by another chain, as no two chains operate at co-multiple rates, the signal can be recovered on the next frame (see figure 10.2).
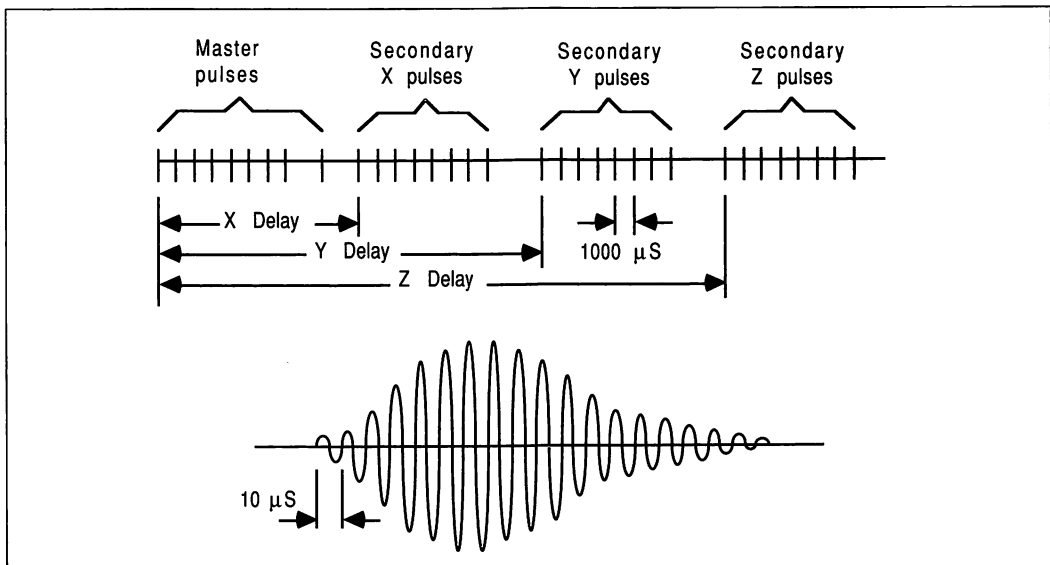
Figure 10.2 LORAN signal format

The transputer has an inherent response time to external stimuli of between one and three micro-seconds, and has an internal resolution of one microsecond. Therefore it could theoretically resolve RF information to an accuracy of three (... (3−1)+1...) microseconds, which at the speed of light would give a navigation system an accuracy of around a kilometre. However, each result can be produced from an average of around 150 such measurements, which would improve the accuracy to around 300 metres. This is because the variable response time can be averaged out, but the resolution cannot because of the high stability of the clocks used.

The design used here improves on this accuracy by capturing the phase of the incoming signal relative to a crystal clock. The amplified filtered signal is used to clock a latch to sample a counter. In keeping with the transputer architecture, the latch used is a link adapter, which allows DMA type tranfer of the data into the transputer, and the appropriate stimulus to the hardware scheduler is generated automatically. The crystal oscillator is inexpensive, as it is required for the transputer anyway, but improves the resolution from one microsecond to better than 10 nanoseconds, which makes analogue noise the predominant problem.

At 100kHz, the events must be trapped at a 10 microsecond rate, which makes the transputer's low latency and rapid process switch time of paramount importance in this application. Any other processor attempting this task would have to hang mid-cycle awaiting the signal on a wait pin to achieve low enough latency, and thus would be unable to perform the trig operations or the system control at the same time. The closest alternative appears to be the Intel 8096, which has a latency of up to 21 microseconds, but does have timers and a fifo that would allow this to be performed only once per seven inputs. This however prevents external upgrading of the internal two-microsecond timer as shown on the transputer based design.

A block diagram of the system is shown in figure 10.3, and a circuit diagram of the digital section in figure 10.4.
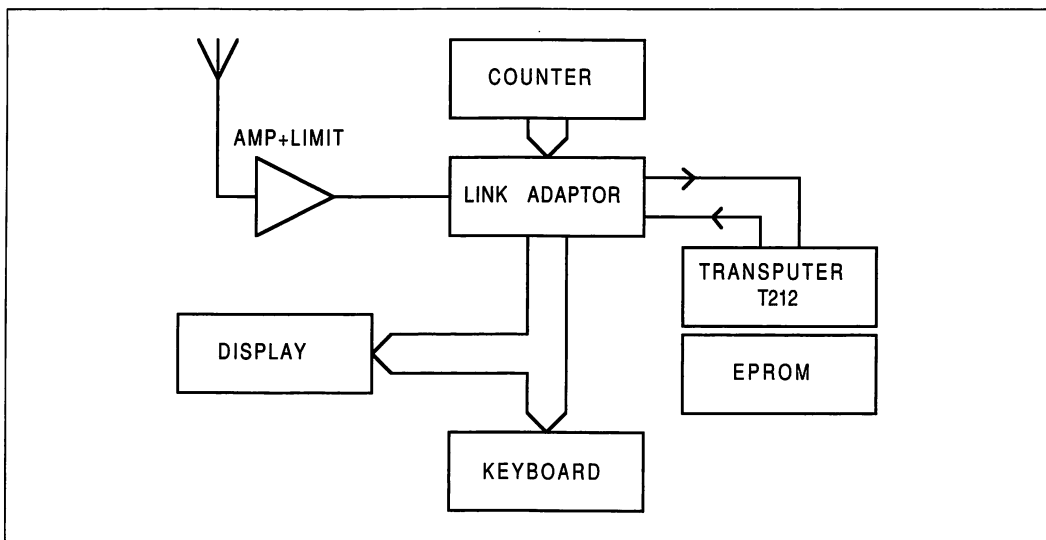
Figure 10.3 Navigation system block diagram
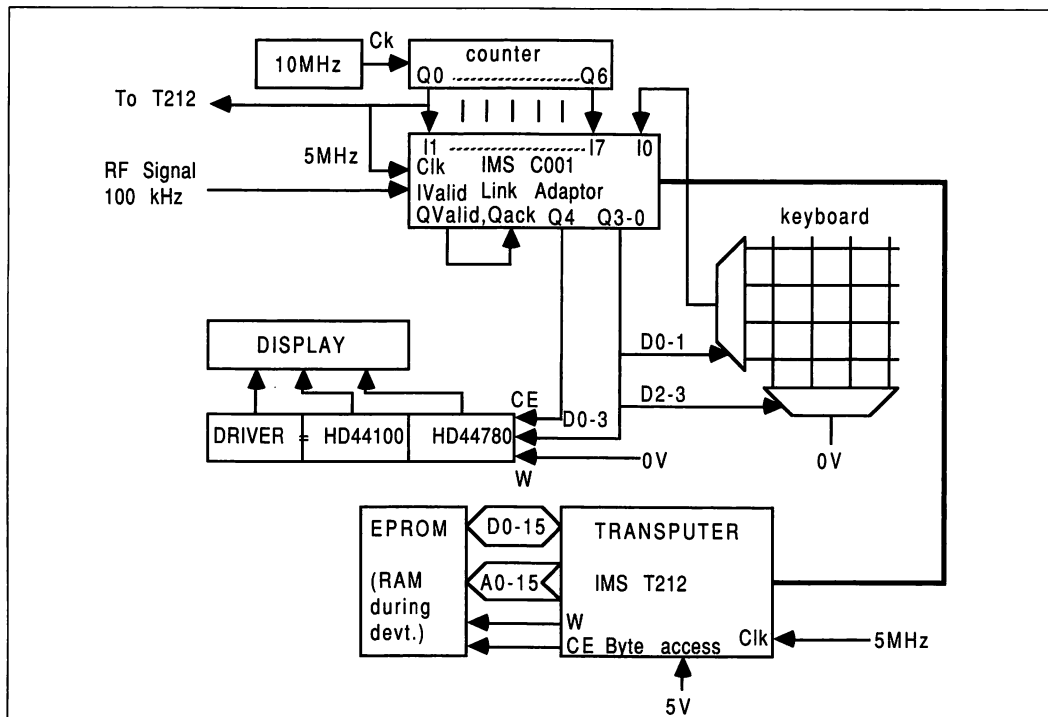
Figure 10.4 Digital circuitry

The basic elements are a high-gain narrow band amplifier to capture the incoming signal, which is only short bursts of RF, so energy-storing LC circuits are avoided where possible, a counter to measure the phase, a transputer to do signal processing and trigonometric calculation, besides controlling the system, and a keyboard and display.

10.3   The I/O system

The interface between the transputer and the analogue and I/O sections has been designed using the IMS C001 Link Adapter. The total input requirements are one bit from the analogue section, the carrier; seven from the counter, and one from the keyboard scanner. The carrier is not fed to the transputer, but used as the 'Input Valid' signal to strobe the current value of the counter, i.e. the relative phase of the signal, into the link adapter input pins. The spare input, D0, is used to receive the open/closed signal from the keyboard matrix, if implemented.

The output requirements are four bits to drive the keyboard scanner, the same four being used as a data bus to the LCD display driver, and a bit to clock the display driver. This leaves three bits spare, which could be used to enlarge the keyboard matrix, or to operate LEDs or alarms for carrier fail or offtrack error.

The keyboard scanner works by taking the four bits D0-3 and decoding them in a CMOS analogue multiplexer into two one-out-of-four signals, giving a 16 key crosspoint matrix. The appropriate Y-wire is connected to ground, and the selected X-wire is fed to the D0 input of the Link Adapter, with a pull-up resistor. Thus, if the currently scanned key is depressed, the input will be a zero, if not, it will be a one. The processor must scan the keyboard, by outputting all 16 possibilities on the highway, at an appropriate interval, say 100 milliseconds. There are three further bits available on the output side of the link adapter that could be used to expand the matrix to 64 keys, which would allow a QWERTY keyboard in a more sophisticated implementation.

The display is an LCD module, available assembled complete with controller, or buildable separately. The driver is a Hitachi 44780, and this is configured to communicate in a four bit mode, so that it can be driven from a link adapter, with a fifth bit used as a timing strobe under software control.

10.4   The processor

The processing module is entirely separate from the rest, and this may be useful in improving the screening of the sensitive analogue stages from processor noise.

The IMS T212 transputer is a self contained computing engine, with RAM, CPU, timers, communications controllers all on the one chip. The only external requirements in this type of embedded system are a program ROM and a 5MHz clock.

The ROM is a standard EPROM, as fast as possible. If an EPROM is available that can keep up with the transputer (100ns cycle!), then no TTL is required, all the necessary chip enable signals are generated by the transputer. Choosing a slower speed option transputer may be worthwhile, for this reason alone, providing the faster parts of the software still operate. If higher performance is required, choose a fast option transputer and use a shift register off the transputer clock to delay memory cycles using the wait pin. If this solution is chosen, there are benefits in copying the code for the front end signal capture process into internal RAM at start-up, giving the ultimate performance.

Only one ROM chip is required, as the T212 has the ability to use 8 or 16-bit data paths externally, depending on the state of an external pin.

10.5   The software

The occam language gives the programmer the ability to map his application onto a yet-to-be determined number of processors, maintaining all the potential for parallelism that exists in the underlying application. The methodology of a parallel language is very different from the sequential approach. At the highest level, all the requirements of the system can be specified as inputs and outputs to a monolithic process, and the specification of that process is the relationships of its outputs to its inputs. Thus we can draw the

diagram of figure 10.5. However, processes are hierarchical, that is we can divide up the work of the main process into several subsidiary processes, with appropriate interconnections, and similarly specify each of them individually. They do not interact in any way except by messages over the connecting channels, as there is no shared memory, so each can be separately debugged, and the decision as to which is in hardware, and which groups on which processor can be left until later. This divide-and-conquer mechanism can be repeated indefinitely, until the base processes are simple to write and thus error-free (see figure 10.6).
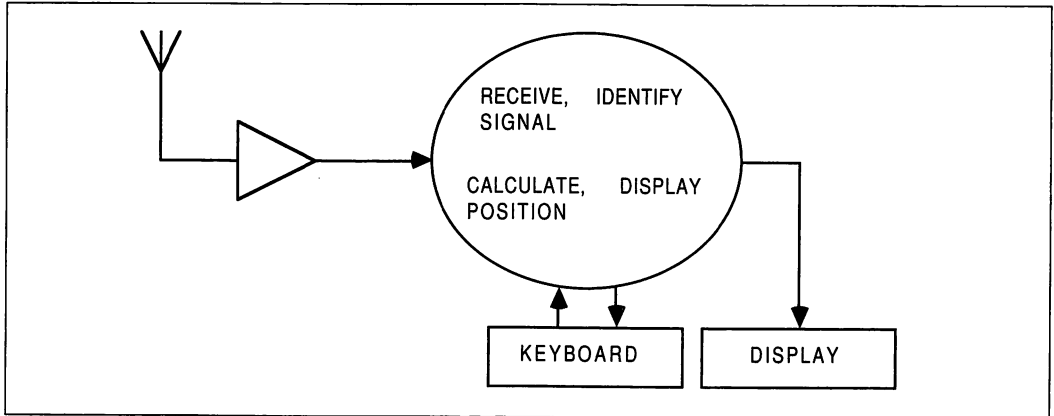
Figure 10.5 Overall function process diagram
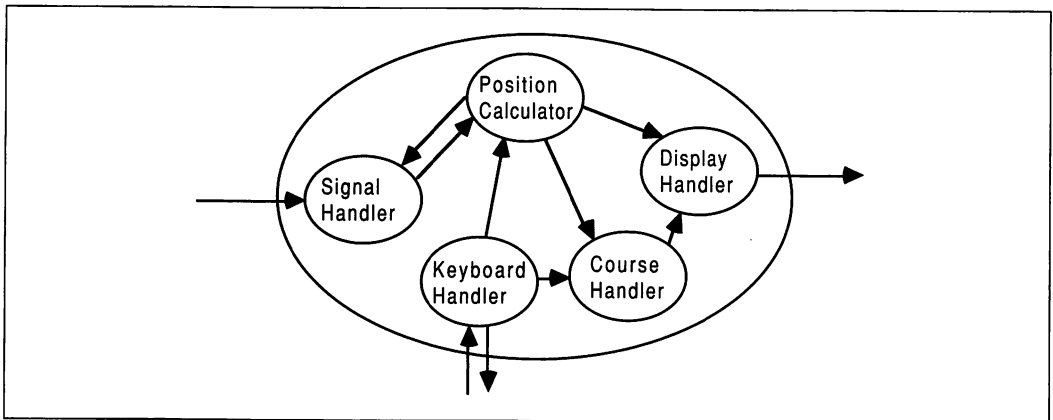

```
CHAN OF INT signal, keyin, keyout, displayout:
PLACED PAR
   PROCESSOR 1 T212
   CHAN OF INT s.to.p, p.to.c, p.to.s, k.to.c, c.to.d, p.to.d:
   PLACE signal AT link0in:
   PLACE displayout AT lionk0out:
   PAR
      ...   Signal handler        (s)
      ...   Position calculator   (p)
      ...   Keyboard handler      (k)
      ...   Course handler        (c)
      ...   Display handler       (d)
```

Figure 10.6 Detail Process diagram and top level occam

For the lowest processes, which are sequential, normal flowchart practice can be used, although with correct system design and well commented code, the programmer can go directly from the process diagram and spec to the occam source. The signal processing function divides cleanly into three processes as shown in figure 10.7. The first process identifies valid carrier transitions, the second valid carrier bursts, collating them into groups, and the third identifies the elementds of the required frame, corresponding with the group repetion interval of the LORAN chain in use.
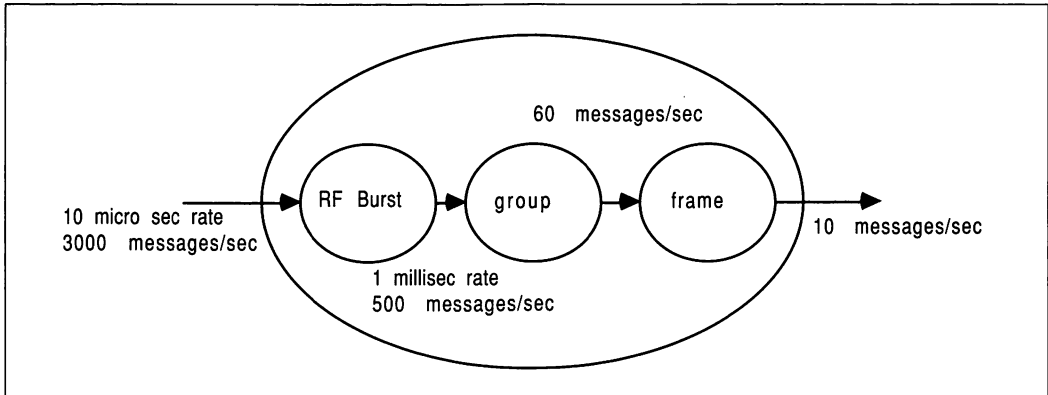

```
{{{  navsys
PROC signal.processing (CHAN OF INT keyboard, screen)
...  proc decls

  ...  decls

  PAR
    ...  test harness

    rfburst (phase.in, burst)
    group (burst, group.element)
    frame (group.element, delays.out, control [chains + 1])

:
}}}
```

Figure 10.7 Sub processes for signal processing

```
{{{  declaration of proc frame
PROC frame (CHAN OF INT in, out, control)
   ... decls and defs

  SEQ
     ... new GRI if offered

     ... START-UP

     ... debug

     ... RUN

:
}}}

{{{  RUN
SEQ i = 0 FOR 4
  missed [i] := 1
count := 0
in ? type; phase; time
WHILE running
  SEQ
     ... new GRI if offered

     ... COMMENT debug
     IF
       NOT (time AFTER (grouptime [count] MINUS margin))
         in ? type; phase; time  --replace as too early
       NOT (time AFTER (grouptime [count] PLUS margin))
         ... correct, pass on after noting
       TRUE
         ... missed some signals
}}}
```

Figure 10.8 occam for signal detection, overview and detail

10.6   Position calculation

The RF signal, suitably processed, gives the difference in distance of the receiver from the master and slave one, and the master and slave two. No absolute distances are known, only the two differences. Simple systems present these differences on a display, and the user must look up two sets of lines on a special chart, locating himself at the intersection of the two lines.

The transputer has enough number-crunching ability to solve the complex trigonometry to calculate the position directly. This is a very difficult calculation, as roughly it is the intersection of two hyperboloids (the distance differences) and a sphere (the earth). However, the hyperboloids are not true mathematical ones, as the generators were not straight-line distances, but great circle routes over the surface of the earth.

This problem does not arise on the short range navigators, because the surface of interest approximates to a plane. In the LORAN system, three approaches are possible. One can assume a position and iterate from it until an accurate result is found. The problem with this is making the program sophisticated enough to detect when the solution will not converge. A second method is to assume linear transmission paths, calculate a rough position, correct the distances and recalculate, repeating until the desired accuracy is reached.

The third and most desirable solution is an analytical one, so the transputer simply calculates some equations. The calculation requires about twenty trig operations, including inverse operations, with a few squares and square roots, and the transputer can easily calculate this to update the position every transmission frame. This is probably not desirable, as it may result in unacceptable jitter in the least significant displayed digit,

so the solution is to only re-calculate and display the position after a set of differences has stabilised. A 10 second update suffices, as this only reduces the accuracy on high-speed powerboats, and at 50 knots, one covers about 300 metres in that time.

10.7   System integration

Once the software is written and tested on the development system, using a dummy occam process, or harness, to feed in phase values and keyboard operations, and capture display results, the code is downloaded into a complete prototype. Using a RAM in the EPROM socket, the code can be changed at will from the development system keyboard at an occam level, with the system operating full speed off air to its own display, with or without additional monitoring information being sent up to the development system.

10.8   Conclusions

The design has shown that the transputer's speed allows functions normally performed in hardware to be brought into the processor, with gains in both assembly cost and flexibility. It has shown how an application may be rapidly taken from the concept to pre-production phase due to the ability to run the prototype attached to the development system, giving the manufacturer a time advantage in the market-place, and a product can be maintained, updated and extended at any time often by issuing only new software.

11 The transputer based navigation system – an example of testing embedded systems

11.1 Introduction

This note covers the implementation of the Navigation System outlined in Technical Note 0, *'A transputer based radio-navigation system'*.

The software described in Technical Note 0 consisted of 4 concurrent processes in a pipeline, as shown in figure 11.1.
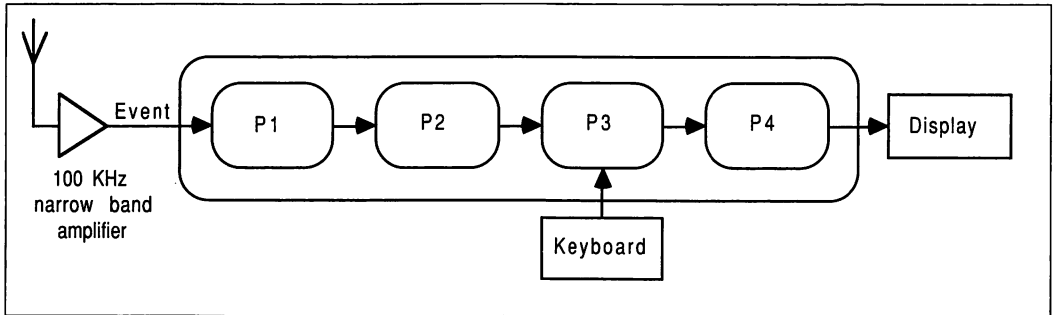
Figure 11.1

These processes performed the following tasks:

P1 Burst detection
P2 Group detection
P3 Frame detection
P4 Position calculation

Just as the 'Divide and Conquer' method eased the design of the software, similarly it allows the software to be tested and debugged without difficulty.

Each process is provided with input data, and its output is checked. Taking the independence of each process into full account allows independent test-data generators to be produced for each, and this is the recommended method if **P1** thru **P4** are being developed simultaneously by separate teams. However, when one team is developing each in turn, only a single test generator is required; when **P1** is correct, its output can be used to test **P2** and so on. Note that this latter method does not test the resilience of subsequent processes to incorrect data, while the former method does.

The system does require resilience to incorrect input data, even if **P2** to **P4** do not and the method of ensuring this is covered later.

Once the code for **P1** is written, a test-data generator is required. This software test-data generator replaces the hardware environment that would normally feed the data.

The most convenient way of testing is to ensure that the process accepts correct data first, and then to extend it to correctly reject erroneous data. To generate the correct data, another process is written.

In the case of the navigation system, the input data is the off-air signal from a chain of transmitters. The incorrect data is interference from other chains of transmitters and from random noise. Thus the first test harness consists of a control environment that manages keyboard and screen of the development system, and a process that mimics a chain of transmitters on figure 11.2.
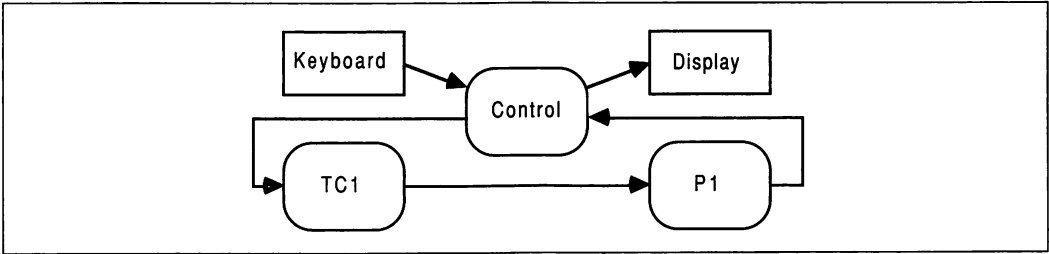
Figure 11.2

This would be ideal, but when it is wrong, how can an error in the controller, **TC1** or **P1** be traced? In this case the harness is debugged by first using just **TC1** with the control — figure 11.3.
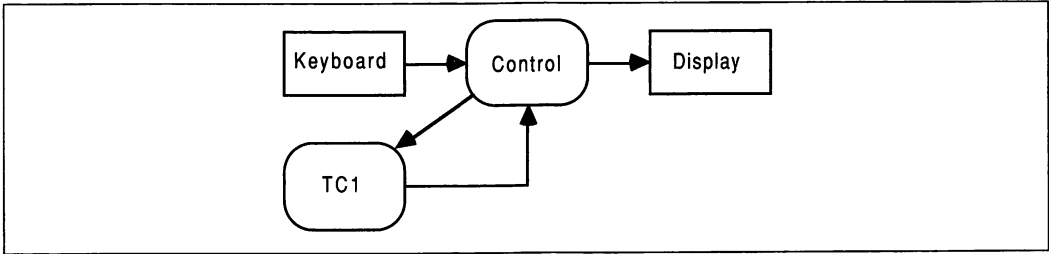
Figure 11.3

This allows **TC1** and the controller to be interactively tested on-screen; feeding in new parameters and checking the data generated.

The generated data consists of a stream of numbers, being the timestamp associated with each zero-crossing of the carrier waveform. The carrier is in groups of bursts, as shown below in figure 11.4.
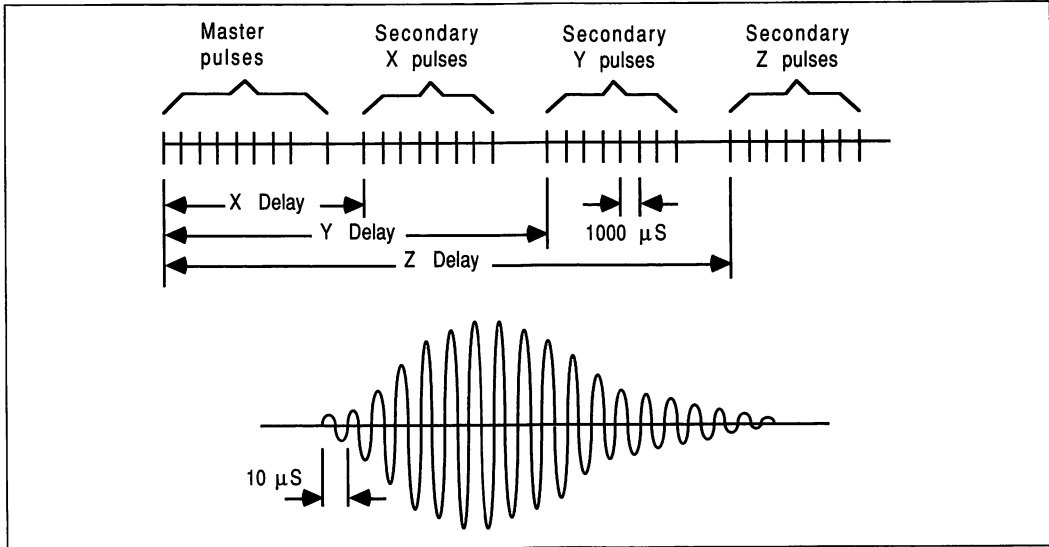
Figure 11.4

The parameters fed to **TC1** are Delay 1, Delay 2, Delay 3 and the Group Repetiton Interval (GRI). In order to facilitate testing, the development system screen was divided into 3 windows, and a menu created. The menu controlled the test environment, displayed in the first window, and the user inputs to the navigation system; i.e. its front panel controls were displayed in the second window. The third window displayed the results from the system, and so represented the front panel display of the navigation system.

11.2   Testing the burst detector

Once the harness was debugged, the configuration of figure 11.2 was used to debug and tune **P1**. 'Tune' should be stressed because there were many constant parameters to each process that determined how selective/tolerant it should be, there being a trade-off, of course, between tolerance, accuracy, and resilience; defined here as the ability to continue functioning in the face of adverse conditions - for example in the case of intermittent lack of input data.

The job of **P1** is to monitor each supposed carrier transition, validate it as being the correct frequency, and of adequate duration, then pass on its initial timestamp and mean phase to **P2**.

As the incoming carrier has a frequency of 100KHz, consecutive events should occur at 10 microsecond intervals. Thus **P1** checks that the interval is within limits (currently set to 9 to 11, as the system implemented differs from Technical Note 0 in feeding the signal direct to the transputer's event pin, giving 1 microsecond resolution on the internal timer, rather than via an external timer).

It then counts a preset number of validated transitions, and if it reaches the threshold, currently set to 10, it accepts the signal as being genuine and passes on to **P2** a timestamp-pair, consisting of the timer value of the first transition and the sum of the 10 phase values. This latter figure allows the effective resolution to be increased by a vernier effect between the RF carrier and the transputer crystal over the whole burst, or group of bursts.

P1 was tested and tuned until the bursts of signal at its input were correctly presented to **P2**; or at this stage, displayed on the screen.

One of the functions of **P1** is to discriminate against noise, so to test this the ability to inject noise was required. This was achieved by expanding the test harness to generate noise. This meant two new processes, one to generate timestamps representing noise, and the other to multiplex the data sources, sorting timestamps into the correct order — see figure 11.5.
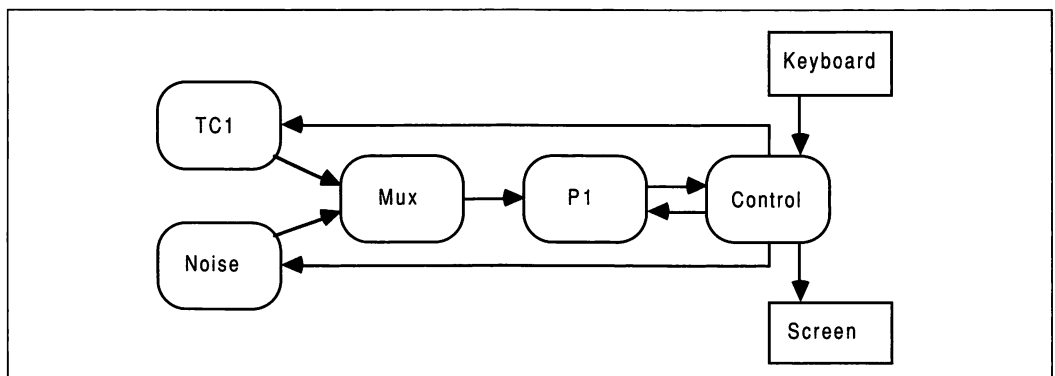
Figure 11.5

Although not fully rigorous, the noise type chosen was bursts of carrier described by their carrier period, the number of cycles in a burst, and the burst repetition rate, so each of these became parameters in the menu window.

The multiplexer simply performed an input as necessary on each stream to ensure it had access to the next data item on each stream. It then selected the earliest timestamp, and passed it to **P1**, replenishing itself from the stream chosen. Notice that no analogue level was considered - the high gain limiting amplifier was considered to have made all inputs full strength. However, time distortion was added; if two timestamps were too close (currently 4 microseconds), they would both be deleted, and replaced with a single transition at the mean of the two: - again, not rigorous, but implementing some approximation to real interference.

11.3   Testing the group detector

Once **P1** had been proven to the harness, **P2** was added. The function of **P2** is to monitor the carrier bursts it receives, and validate them into correct groups for master or slave transmitters. A slave transmitter generates eight bursts at one millisecond intervals, and a master 9 bursts, spaced as if the group were ten bursts with the ninth omitted.

It can be seen that there is massive data reduction down the pipeline. **P1** expects an input every 10 μs, **P2** every 1 ms, **P3** approximately every tenth of a second; these are peak rates - the duty cycle is very low. As a result of the data reduction, more thorough testing is feasible as the later processes are added, as the volume of data on the screen reduces.

This implementation uses visual checking; it would be perfectly possible to correlate output and input in another process and report only statistics. This method was rejected because the final navigation system generates only two outputs - LATitude and LONGitude; the visual approach is entirely satisfactory.

To validate bursts, **P2** checks that they are at one millisecond intervals, plus/minus a tolerance, currently set to 5 microseconds. Again, the benefit of the harness is seen in allowing the system to be tuned. It then counts validated bursts. The subtle part is how to optimally detect master transmitters, as the process only runs when triggered by an input, so if the final pulse never comes, it is a slave, but the process does not run to report this.

The solution is simple, once found. It is important not to waste CPU time, so to deschedule the process and wait on a timer for 2+ milliseconds would be a problem, but is the easiest to implement. However, there is no problem of latency in the pipeline - it does not matter if the screen display runs milliseconds after the input - all the data inputs were timestamped on reception, so accuracy is maintained. Thus no output is generated until the next input burst, when the decision is made whether it is the ninth burst of the group (i.e. it was a master) or the first of an independent group (it was a slave).

Part of the validation task performed by **P2** is to reject groups that have been corrupted by overlapping between two transmitter chains.

If the bursts collide directly, **P1** will reject them. However, because of the low duty cycle it is possible that they may interleave. In this case the current implementation of **P2** will lock onto the group starting first, and ignore the interleaved bursts as each is 'too early' in its opinion. This is not the optimum solution, as the second group may be the desired one. However, **P2** is ignorant of this, it being decided in **P3**, and to track two groups simultaneously adds unnecessary complication. It could be done, however, if the LORAN time domain became too cluttered in some areas.

All these functions can be tested by adding a second transmitter chain (TC2) to the environment. Experiments can then be performed with the two chains with very close repitition intervals. Again, due to the data reduction, this testing can be extended greatly after **P3** is written.

The final test harness is shown in figure 11.6, used first with **P1** and **P2**, then **P1** to **P3**, then **P1** to **P4**.
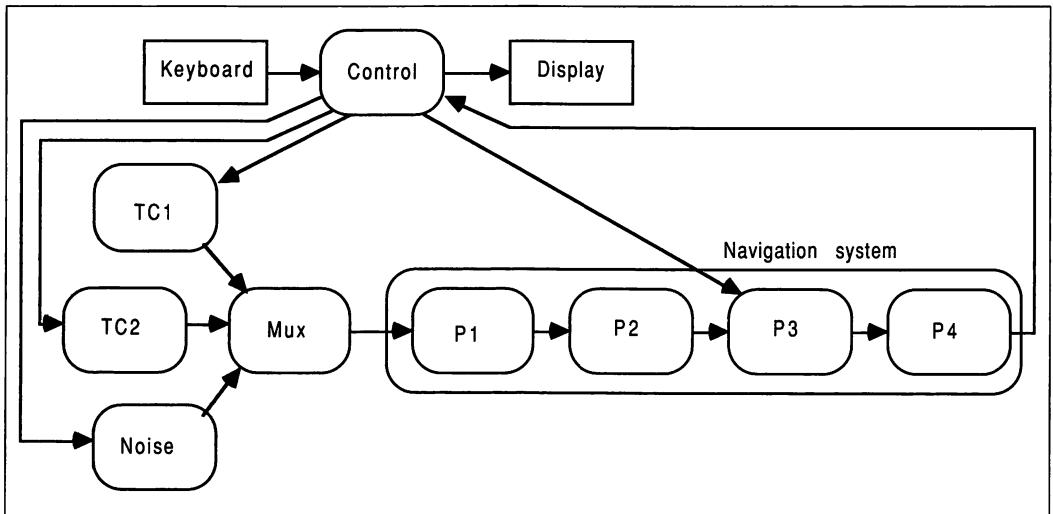
Figure 11.6

11.4   Testing the frame detector

P3 is the most complex and thus requires most testing and tuning. Its task is twofold — i.e. it has two modes of operation. First it must identify and lock onto the correct transmitter chain, then it must monitor it, even though a large percentage of its transmissions may have been lost due to noise or other transmitters interfering.

The first task is performed by capturing a buffer full of detected groups, and then searching the buffer for groups that have the correct repetition interval. The buffer must be large enough to cover at least two frames, in order that spurious internal matches be excluded, and again, the tolerance on the matching requires tuning.

If there is not suitable match, the initialisation phase starts again, and repeats until successful.

Once the timestamps of the required transmitter chain are found, the process predicts when the next will be, and validates against that. If a timestamp is missed, a new prediction is made, and the omission noted. After a set number of omissions in a row (currently 5), the system admits a synchronisation failure and reverts to initialisation mode.

Thus the 'locking' criteria can be tuned against the 'unlocked' criteria. As set at present, there will be the occasional false lock, which will then find no valid frames and re-initialise. Final tuning of this will be done in the real world, when the level of noise etc. is real, not simulated.

At each successful frame, **P3** passes on the delay values to **P4**, which performs the mathematics and displays the ship's position.

11.5   Improvements during testing

Two improvements were made to **P3**, **P4** to maximise the performance of the system.

In **P3**, allowance was made for errors in frequency between the transmitter crystal and the transputer crystal. Although partly covered by the timing tolerances in **P1** to **P3** already, because **P3** assumes missed signals, and predicts future ones, any error is multiplied by the number of frames covered. Thus while it is instructed to use a particular Group Repetition Interval, it will actually use one extracted off-air, within a tolerance (currently 48 microseconds).

This greatly improved the system noise tolerance.

In **P4**, rather than update the display every tenth of a second, which is too fast for the human eye, causes excessive least-significant digit jitter, and uses excessive CPU time, the delay signals were validated by collecting them for a period (currently 2 seconds), rejecting jitter-rogues, and then calculating and displaying.

11.6   Conclusions

It can be seen that the software harness allowed demonstration of the system, basic debugging, error-handling, performance enhancements, all before an oscilloscope was bought to test the hardware! It will also allow continued testing with real input data, but display via the development system, giving the opportunity for final program tuning in RAM before the ROMs are programmed and the system goes live across the ocean.

12 A transputer based distributed graphics display

12.1 Introduction

This technical note examines a frame store distribution technique using the IMS T800 for high performance computer graphics systems.

Firstly there is a brief introduction to some of the techniques and terminology used in typical graphic systems including comments on system implementation and processing implications.

Following this, section 12.3 provides an overview of parallel graphics systems and frame store distribution. There is also brief descriptions of the transputer, specifically the IMS T800 architecture, the occam language and transputer module architecture. Following this there is an introduction to the two TRAMs used to implement the distributed graphics system.

The next two sections describe the graphics TRAMs in detail, and how the distribution methods are implemented.

Finally some example system configurations are described using the graphics TRAMs and some performance implications of the configurations.

12.2 A brief history

12.2.1 Introduction

In the early days of computing, user interaction with computers usually consisted of a teletype machine with a built in keyboard. This was costly in terms of maintaining the mechanics and producing reams of partially used paper. It wasn't long before electronic displays began to be commonly used. The first displays were essentially *glass teletypes*, providing the user with an electronic alphanumeric display. The visual display was constructed from a two dimensional array of dots called **pixels**. Each pixel had one colour and could be illuminated individually -either on or off, hence the name **monochrome** (monochromatic) display. From this any character could be represented provided it was constructed from a small array of dots that fitted into one character matrix size on the screen. Since then these displays have become more sophisticated, having large numbers of displayable colours and higher numbers of unique displayable dots per square unit of the screen surface.

12.2.2 Displays

Most electronic displays consist of an evacuated sealed glass tube, with a coating on the inside surface of the display screen. A beam of electrons are fired onto the coating, which makes it glow, producing a small spot of light. Because the beam is moving charge, it can be deflected using either electrostatic or magnetic fields. Its intensity can also be controlled, changing the brightness of the spot. This allows the path of the spot and its brightness to be controlled by electronic circuitry (see figure 12.1).

These circuits are designed to make the beam scan in a series of horizontal sweeps, left to right across the display. When the beam reaches the end of the line, it's brightness will be switched off (blanked) and it will fly back at high speed to the start of the next line, slightly below the previous line. This is known as **line flyback** (see figure 12.2). This scanning will continue until the entire display has been scanned. When the beam reaches the end of the last line it will be blanked and will fly back at high speed to the top of the display, This is known as **frame flyback** (see figure 12.1). This happens so fast that the human eye cannot see the spot, and the lines are so close together that they are not individually perceivable at normal viewing distances. A small spot of light can produce a complete frame so fast that it can be animated without being perceived as individual frames. This is a similar technique to that of the film industry, where multiple still frames give the illusion of a moving picture.

Some systems use a technique known as **interlace**. Each frame of a scene is split into two **fields**. Each field contains every other line of the complete frame. So one field contains all the odd numbered lines and the the other all the even lines. This technique allows each field to be displayed for the same period as a complete frame, without causing much of a flickering effect. This halves the rate of data that needs to be displayed, reducing the necessary speed of the electronics. Television systems use this technique to reduce
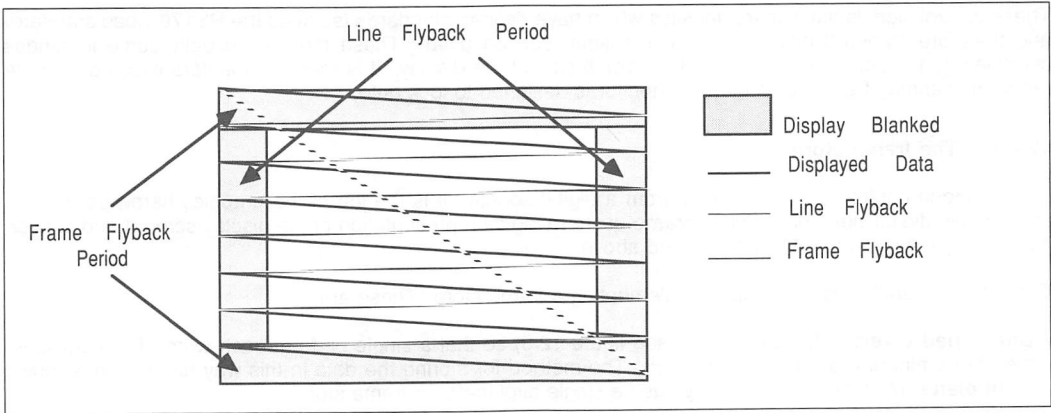
Figure 12.1 Display scanning

the bandwidth of the transmitted signal.

The circuitry controlling the horizontal and vertical scanning frequencies of the beam and the brightness of the spot can be controlled using an input control signal. This control signal is continuously variable in the range of 0 to 1 volt. The brightness of the spot is represented by the input signal voltage level in the range 0.3 to 1 volt. Synchronisation pulses (pulses that control the frequency of the scanning spot) are represented by the control input signal voltage level in the range 0 to 0.3 volt (see figure 12.2). The synchronisation pulses are superimposed onto this signal by the graphics hardware, so that the display scanning circuitry will scan in lockstep to the scanning of the frame store. This ensures that the data representing a particular pixel on the display will always be at the same place on the screen (see figure 12.2).
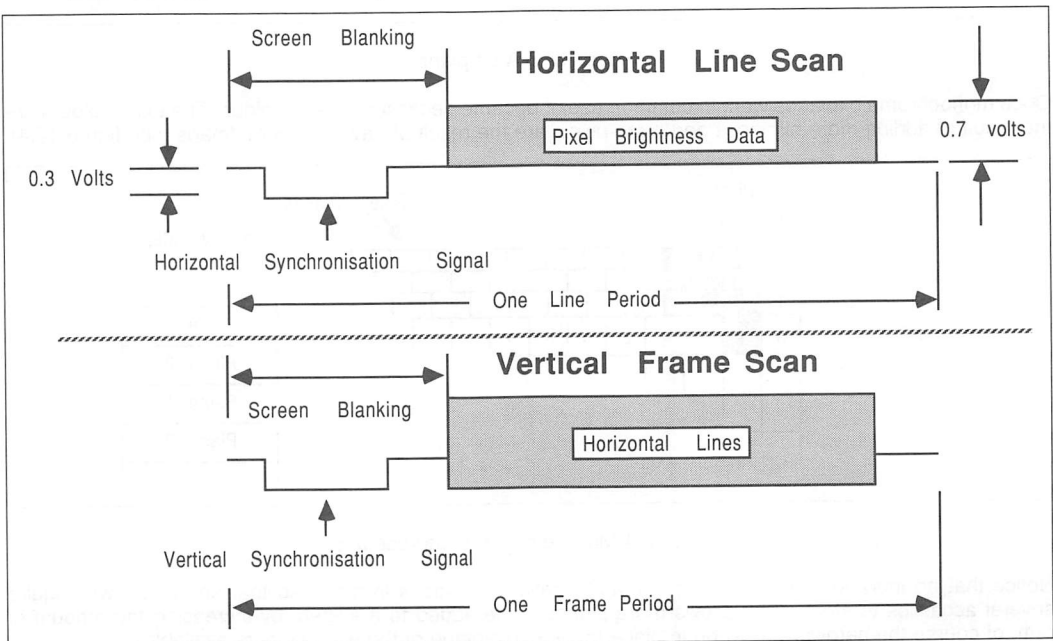
Figure 12.2 Analogue control voltage waveforms

These control signals have characteristics which have defined standards (such as the RS170 video standard) and therefore standard displays, called monitors, can be used. These monitors usually come in ranges classified by the screen dot size and the overall size of the display. It is these two factors which define the range of scanning frequencies that the monitor is designed to lock onto.

12.2.3   The frame store

The analogue control signal is derived from a digital source. It is the job of the graphics hardware to scan and retrieve digital video data from a **frame store** (a digital representation of the display screen) and convert it into the analogue control signal outlined above.

There are generally two methods of implementing a frame store. These are:

• **Bitmapped pixels:**   Data is stored (see figure 12.3) so that a single bit from each word of a processors store will illuminate a pixel either on or off. The method for storing the data in this way has become known as a **bitplane**. Monochromatic displays use a single bitplane as a frame store.
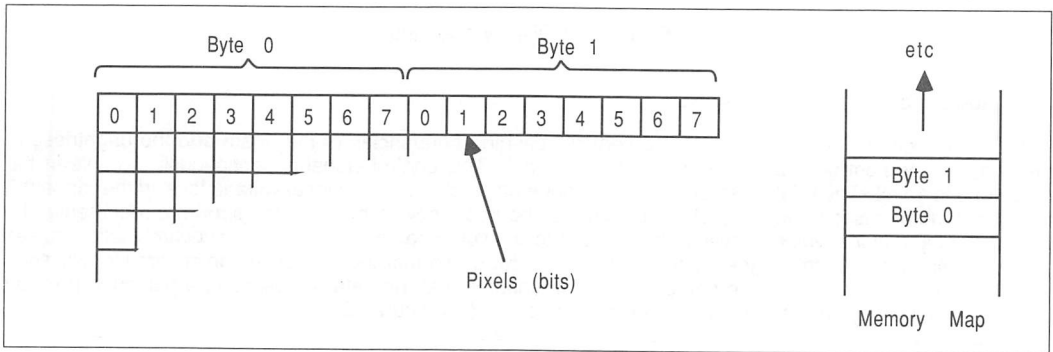
Figure 12.3 A bit plane

Once monochrome bitplanes were in common use, it became necessary to add colour. The extra colours are the result of adding more bitplanes and more pixels are the result of having larger bitmaps (see figure 12.4).
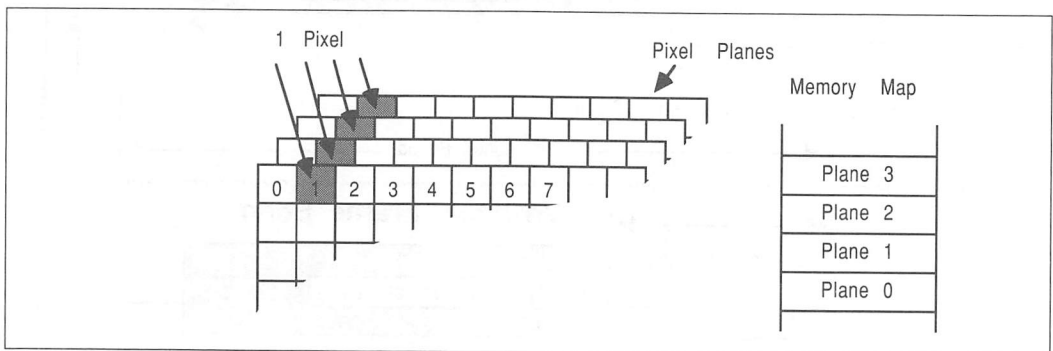
Figure 12.4 Multiple bitplane address map

Notice that an individual pixels data is spread to several locations in store, so that an update will require several accesses to store. This allows more planes to be added to a system by increasing the amount of ram, of course the hardware must be in place to take advantage of the extra colours available.

• **Packed pixels:** Data is stored so that each pixel is located at a single address in store. This provides an efficient memory access utilisation at the cost of fixed numbers of colours per pixel.
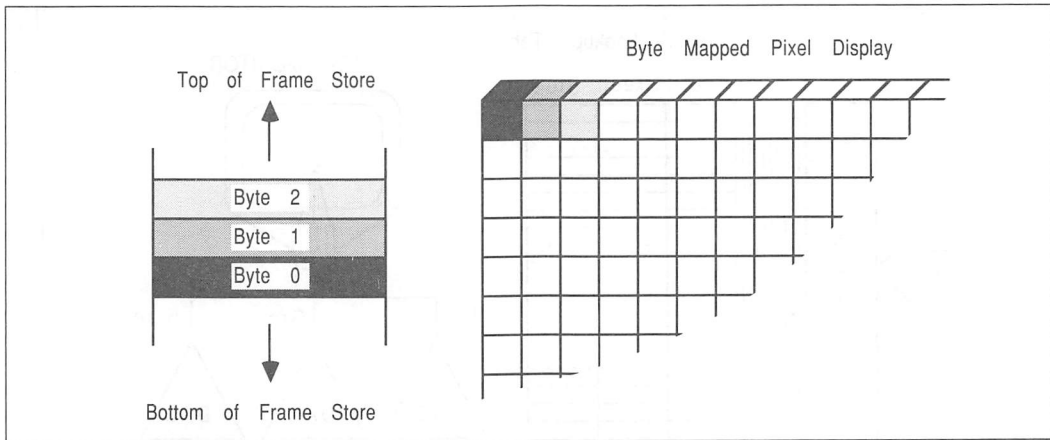
Figure 12.5 Packed pixel organisation

Any frame store implementation must be scanned by hardware continuously so that the the pixel information can be encoded onto the analogue control signal. Also, the frame store must be available for modification by the processor. The hardware must therefore arbitrate the frame store access between the **display scanning** and processing (see figure 12.7).

12.2.4   Colour

Colour monitors use three different colour sub-pixels (as close to the three primary colours, red, green and blue, as possible) that can be illuminated separately. For this, three separate control signals, which vary the brightness of each colour, are necessary.

To produce these colour signals, the digital data is separated into the three colour components red, green and blue. Each is fed into a separate digital to analogue converter (DAC). The analogue signal now consists of the three separate signals representing the primary colours. By varying the digital input to these DACs the voltage levels of each these signals can be changed producing a large number of possible colours on the monitor. This can be extended so that digital pixel data can represent an address in a table which has been preloaded with various colour values for each output DAC (see figure 12.6).

This intermediate Colour lookup table (CLUT) can increase the total number of possible displayable colours. This is because the table width is not related to the addressable entries to the table (see figure 12.6). Each entry can output data to each DAC, presenting more bits to all three DACs than the input pixel data contains. Only a small number of the total displayable colours can be displayed at any one time though (the number of unique addressable entries to the table).

For example (see figure 12.6), the colour table may contain 256 entries, each entry is 18 bits wide, presenting 6 bits of colour value to each DAC. This gives 262144 (2^{18}) possible colour values. Any combination of these colours are allowed since the table is preloadable, but only 256 colours are displayable at any one time.

12.2.5   System performance

In many graphics systems, there are aspects of the design where system performance is reduced, such as in a multiple bitplane addressing (see section 12.2.3). Many systems become **special purpose** to overcome these performance problems and thereby increase the cost of the system by using custom built hardware and reducing flexibility. The following are typical areas where these problems can arise:

• **Pixel addressing:** Each pixel may not have a unique address, ie. when using multiple bit planes. Single bits in many locations in the frame store represent a single pixel, requiring accesses to many locations to
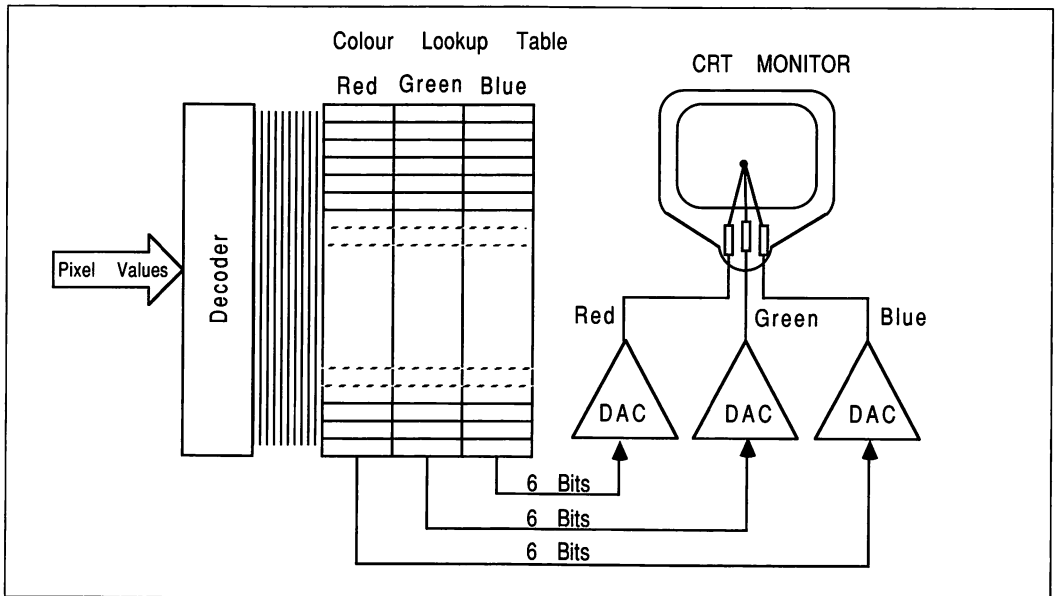
Figure 12.6 Colour lookup table

change this pixel value. General purpose processors do not usually have the ability to manipulate data addressed in this way. Special high speed graphic processors with hardware engines need to be placed between the general purpose processor and the frame store to map pixel data into the frame store (see figure 12.7). These processors come in a range of configurations, ranging from full blown processors with large instruction sets, to a collection of engines designed for highly specific purposes.
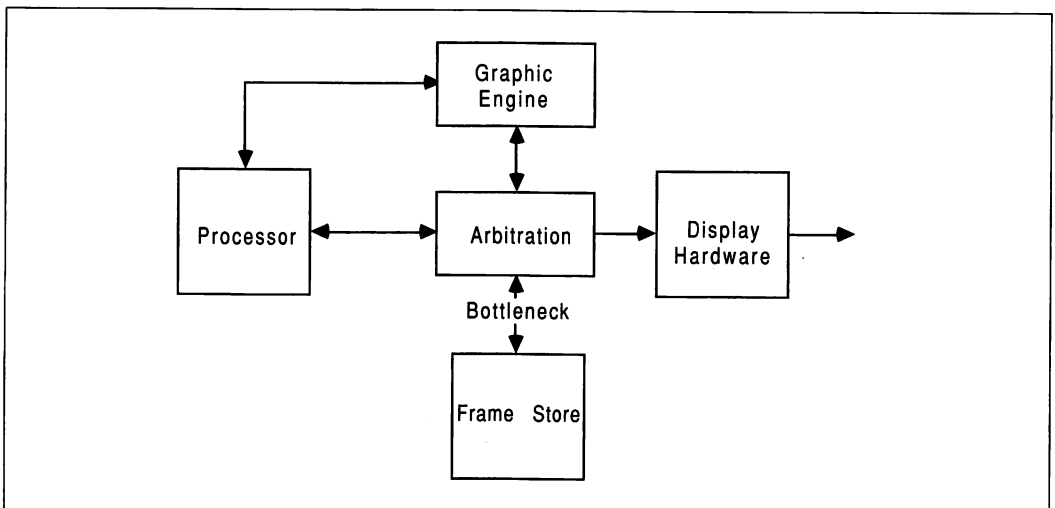
Figure 12.7 Special graphic processor

- **Frame store access conflicts:** The processor must perform drawing tasks into the frame store when the display scanning hardware is not using the frame store. This can consume processor performance because any drawing into the frame store is restricted due to the sheer amount of data that has to be shuffled out of the frame store by the display scanning hardware. This is especially so in high resolution systems. This is referred to as the frame store bottleneck (see figure 12.7).

Consider a 512 by 512 by 8 bit pixel display. If we assume that a 32 bit read from the frame store takes 200×10^{-9} secs., and the store is scanned 50 times a second (20×10^{-3} secs). Then to read all the data will take 65536 reads and will take 13.1×10^{-3} secs. This leaves the processor $(20 \times 10^{-3})-(13.1 \times 10^{-3}) = 6.9 \times 10^{-3}$ secs. to update the display. This leaves only 34% of the total frame store bandwidth for the processor to do anything useful.

Doubling the horizontal and vertical resolution (R) quadruples the frame store data (proportional to R^2). Also, doubling the number of colours (C) will increase frame store access bandwidth. It follows that the processors access to the frame store is proportional to a CR^2 law. This is doubled when we consider that the scanning hardware needs to read all this data as well. This can somewhat be relieved by using several banks of ram and using a *ping-pong* mechanism to switch the busses between the processor and display hardware. This is only useful in animation systems where each frame has to be completely redrawn and therefore becomes somewhat special purpose.

- **Compute performance:** Consider animating a graphic image which consists of 12,000 points (where FLOPs means 'Floating Point Operations').

Operation		Units
Rotate, translate, scale	:300	KFLOPs
Clip (display viewable surfaces)	:72	KFLOPs
Converting to screen coordinates	:130	KFLOPs
Shading	:360	KFLOPs
Interpolation (rounding flat surfaces)	:300	KFLOPs
The approximate total is:	:1.2	MFLOPs

Assuming 25 frames a second, the grand total becomes 30 million FLOPs per second. This level of performance is well beyond single processor performance, indeed just shuffling the data around is beyond memory bus bandwidths of many processors.

12.2.6   Graphics display system

From the above brief discussion, several requirements arise for a **general purpose** graphics system can satisfy the needs described:

- **Compute performance:** Any required compute performance desired for any given application.

- **Drawing performance:** Any required drawing performance into the frame store for a given application.

- **Display access:** The display scanning must have separate access to the frame store to remove the conflict between the processor and the display scanning hardware.

- **Display resolution and colour depth:** Any required display resolutions and colour depth (bits per colour).

- **Display Drivers:** Any required display output (to follow above). For instance, very high speed device technology may be necessary for a very high resolution display.

This technical note will describe a transputer based, distributed graphics system which resolves the problems outlined above.

12.3   Overview of a parallel graphics system

12.3.1   Introduction

In the previous section (section 12.2.6), several aspects of a graphics system were discussed.

To provide any desired processing performance requires that the processing task is divided into smaller subtasks and as many processors that are necessary to provide the appropriate performance must be used. This allows a system to be built to achieve any drawing bandwidth, with any compute performance. The problem is now one of distribution and how this is implemented.

Here are some methods for distributing processing tasks:

• **Spatial:** The display is broken up into a number of tiles. Each tile is distributed to a different processor or a group of processors (see figure 12.8).
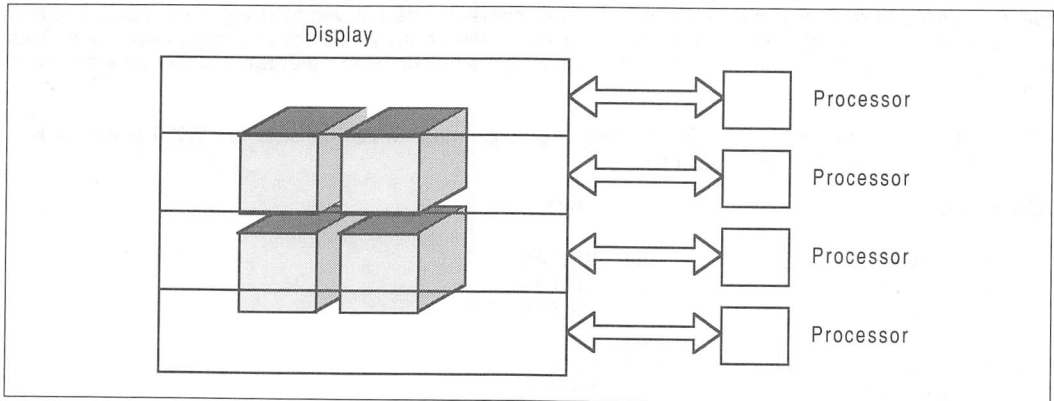
Figure 12.8 Spatial distribution

• **Chronological:** This method distributes the entire display to all processors in the system, but only one will display all it's data at any one time. Each frame of the display is produced by a processor or a group of processors (see figure 12.9).
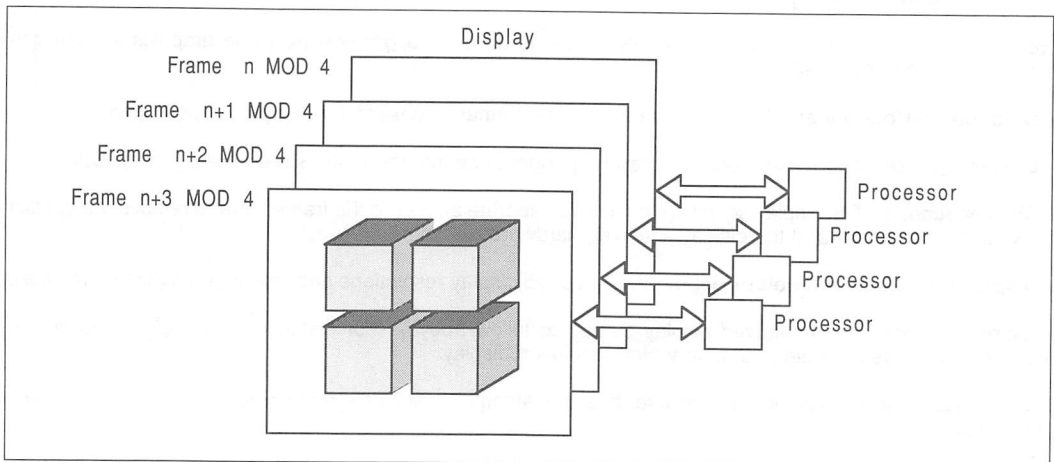
Figure 12.9 Chronological distribution

• **Objective:** This method distributes different objects in a scene to different processors. This is deceptively difficult - consider the problem of handling hidden and intersecting objects (see figure 12.10).
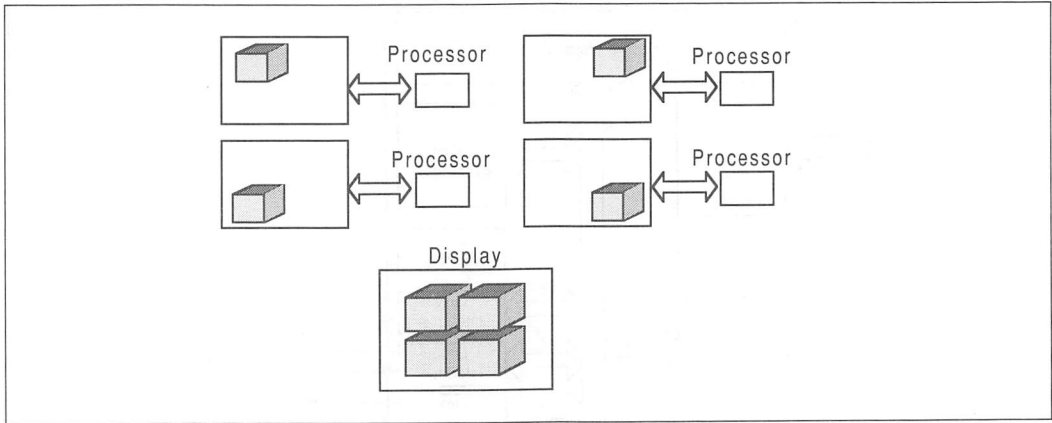
Figure 12.10 Objective distribution

• **Characteristic:** This method distributes characteristics of the scene, such as colour, to different processors (see figure 12.11).
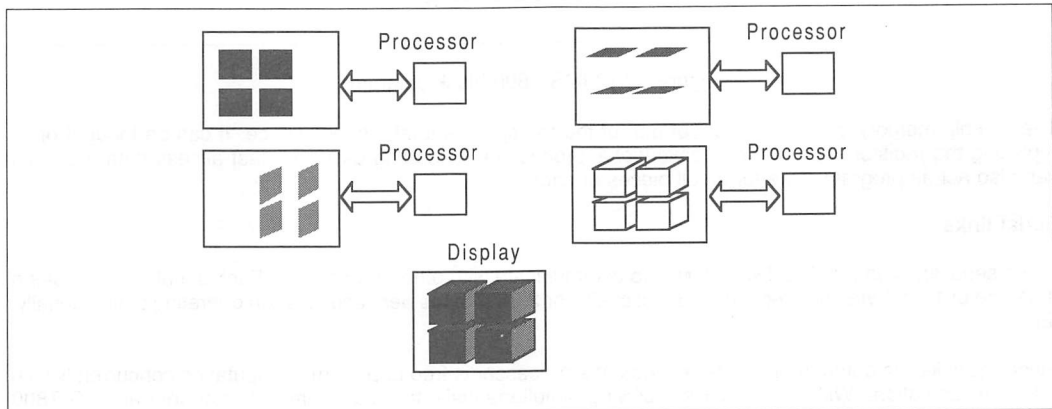
Figure 12.11 Characteristic distribution

These distribution methods are simplified using the occam model of localised data and process communication, applied with the transputer localised processor bus and interprocessor communication.

12.3.2   Transputers and occam

The IMS T800 transputer

The IMS T800 is the latest member of the INMOS transputer family [1]. It integrates a 32 bit 10 MIP processor (CPU), 4 serial communication links, 4 Kbytes of RAM and a floating point unit (FPU) on a single chip. An external memory interface allows access to a total memory of 4 gigabytes (see figure 12.12).

The transputer family has been designed for the efficient implementation of high level language compilers. Transputers can be programmed in sequential languages such as C, PASCAL and FORTRAN (compilers for which are available from INMOS). However the occam language allows the programmer to fully exploit the

facilities for concurrency and communication provided by the transputer architecture.
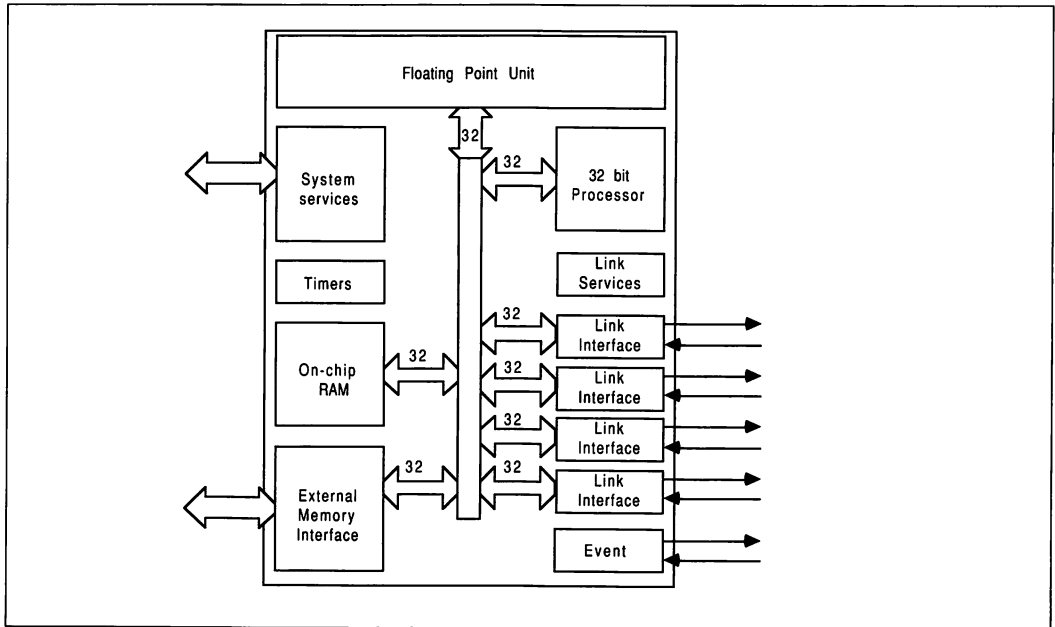
Figure 12.12 IMS T800 block diagram

The on-chip memory is not a cache, but part of the transputer's total address space. It can be thought of as replacing the register set found on conventional processors, operating as a very fast access data area, but can also act as program store for small pieces of code.

Serial links

The 4 serial links on the IMS T800 allow it to communicate with other transputers. Each serial link provides a data rate of 1.7 MBytes per second unidirectionally, or 2.35 MBytes per second when operating bidirectionally, [2].

Since the links are autonomous DMA engines, the processor is free to perform computation concurrently with link communication. With all four links receiving simultaneously, the maximum data rate into an IMS T800 is 6.8 Mbytes per second. This allows a graphics system based around IMS T800s to act as image sinks, accepting pixels down serial links and placing them directly into the frame store.

On-chip floating point unit

The IMS T800 FPU is a co-processor integrated on the same chip as the CPU, and can operate concurrently with the CPU. The FPU performs floating point arithmetic on single and double length (32 and 64 bit) quantities to IEEE 754. The concurrent operation allows floating point computation and address calculation to fully overlap, giving a realistically achievable performance of 1.5 Mflops (4 million Whetstones [3] / second) on the 20 MHz part; 2.25 Mflops (6 million Whetstones / second) at 30 Mhz.

2-D Block move instructions

Among the new instructions in the IMS T800 are those for graphics support. The IMS T800 has a set of microcoded 2-dimensional block move instructions which allows it to perform cut and paste operations on irregularly shaped objects at full memory bandwidth.

The three **MOVE2D** operations are:

MOVE2DALL   which copies an entire area of memory
MOVE2DZERO   which copies only zero bytes
MOVE2DNONZERO which copies only non-zero bytes

The use of these instructions is described more fully elsewhere [2].

The occam programming language

The occam language enables a system to be described as a collection of concurrent processes which communicate with one another, and with the outside world, via communication channels. occam programs are built from three primitive processes:

variable := expression   assign value of expression to variable
channel ? variable   input a value from channel to variable
channel ! expression   output the value of expression to channel

Each occam channel provides a one way communication path between two concurrent processes. Communication is synchronised and unbuffered. The primitive processes can be combined to form constructs which are themselves processes and can be used as components of another construct. Conventional sequential programs can be expressed by combining processes with the sequential constructs **SEQ**, **IF**, **CASE** and **WHILE**.

Concurrent programs are expressed using the parallel construct **PAR**, the alternative construct **ALT** and channel communication. **PAR** is used to run any number of processes in parallel and these can communicate with one another via communication channels. The alternative construct allows a process to wait for input from any number of input channels. Input is taken from the first of these channels to become ready and the associated process is executed. A full definition of the occam language can be found in the occam reference manual [4].

12.3.3   Transputer modules (TRAMs)

Transputer Modules [5] or **TRAMs** are subassemblies of transputers (or other components with INMOS links), a few discrete components, and sometimes some RAM and/or application specific circuitry. All TRAMs:

- Have a standard interface using serial links.

- Have a standard pinout.

- Have standard sizes.

- Are designed to a published specification [5].

These TRAM standards make it very simple for users to build customised TRAMs or motherboards with sockets for TRAMs. The TRAM pinout standard is independent of:

- Transputer type (IMS T212, T414, T800, M212, etc.)

- Number of transputers (0, 1, 4, 8, 16, etc.)

- Wordlength of transputer.

- Speed of transputer.

- Function (transputer plus RAM, disk control, other peripheral control)

- Memory size.

- Package (68 pin PGA, 84 pin PGA, PLCC, and other transputer packages)

- Implementation (PCB, hybrid, silicon, etc)

12.3.4   Introduction to graphics TRAMs

If the graphical display processors are implemented as modular transputer compute elements, each with transputer, memory and logic to implement special functions, the problem of designing a distributed graphics system becomes much simpler.

To provide the distributed frame store requirements and any display output type (see section 12.2.6), two different TRAMs are deemed necessary.

- **Serial port TRAM:** This contains an IMS T800 and all the logic necessary for a complete frame store. It can be connected to other identical TRAMs so that distribution of the frame store becomes a matter of simple replication of this TRAM. This is known as the **Serial port** TRAM because of the serial nature of the output data.

- **Display backend driver TRAM:** This contains all the logic necessary to drive a particular display type. This TRAM interfaces directly to, and receives it's high speed data from, the serial port TRAM. This TRAM will be known as the **Display Backend** TRAM.

Separation of frame store scanning from the processor address and data bus is achieved on the serial port TRAM using video RAMs (see section 12.9). Video RAMs have a separate **serial port** (a port in this context means a separate access path to shared data) for video data. This allows the frame buffer to be scanned in a serial fashion without causing significant loss of processor access to the RAM, relieving the arbitration problems associated with conventional RAMs (see section 12.2.5).

The serial port TRAM supplies a continuous stream of high speed serial data from the frame store. The **Display Backend** can drive display monitors using this stream of data in a variety of display modes. These TRAMs are connected together by a 60 way ribbon cable, which contains a control bus and a distributed data bus. All serial port TRAMs in the system connect in parallel to this cable (see figure 12.13).
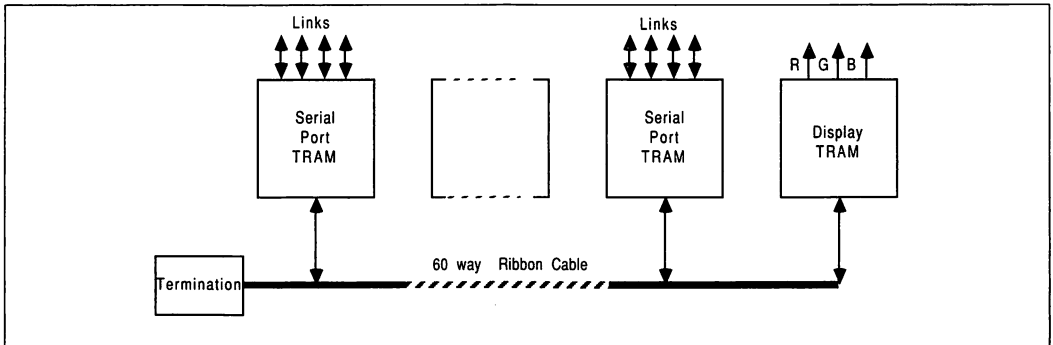
Figure 12.13 Connectivity of graphics TRAMs

12.3.5   An Introduction to the serial port TRAM

This section contains a short introduction to the serial port TRAM. A detailed description can be found in section 12.4.

The serial port TRAM (see figure 12.14) consists of:

- **A transputer:** An IMS T800, which maintains the frame store.

- **Memory:** The standard serial port TRAM contains a total of 2.25 Mbytes of 4 cycle dynamic RAM. Of this 1 Mbyte is standard dynamic RAM and 1.25 Mbytes is Video RAM.
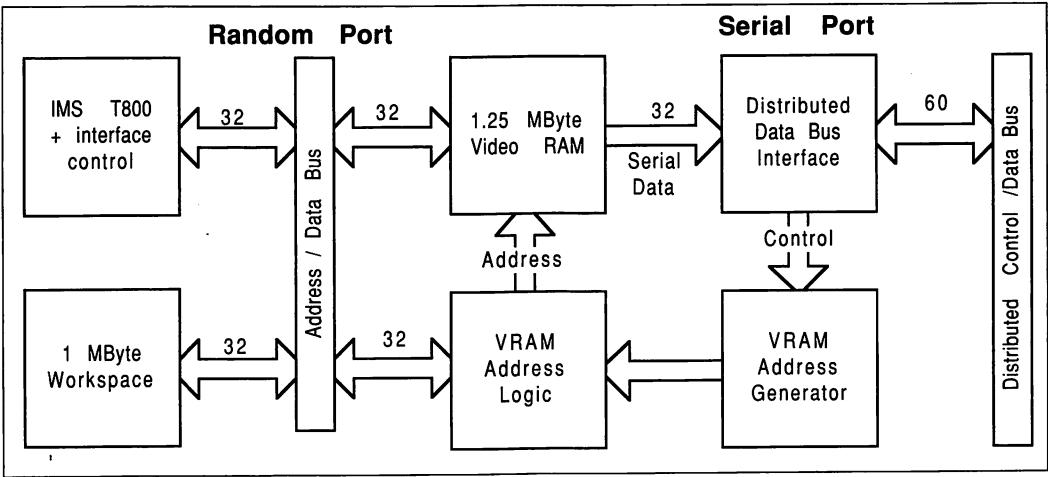
Figure 12.14 Serial port TRAM block diagram

• **Video RAM address generator:** This controls the VRAM serial port addressing. It is in turn controlled by the distributed control bus.

• **Serial bus interface:** This is the distributed serial data and control bus interface. It connects the distributed control bus to the various timing components on the TRAM and the VRAM serial data to the distributed data bus.

Figure 12.14 shows a block diagram of the serial port TRAM, outlining some of the blocks previously described.

12.3.6   An Introduction to the display backend TRAM

All display TRAMs have a generic architecture. Figure12.15 shows the generic block diagram of the display backed TRAM architecture. A detailed description of the Display Backend can be found in section 12.5.
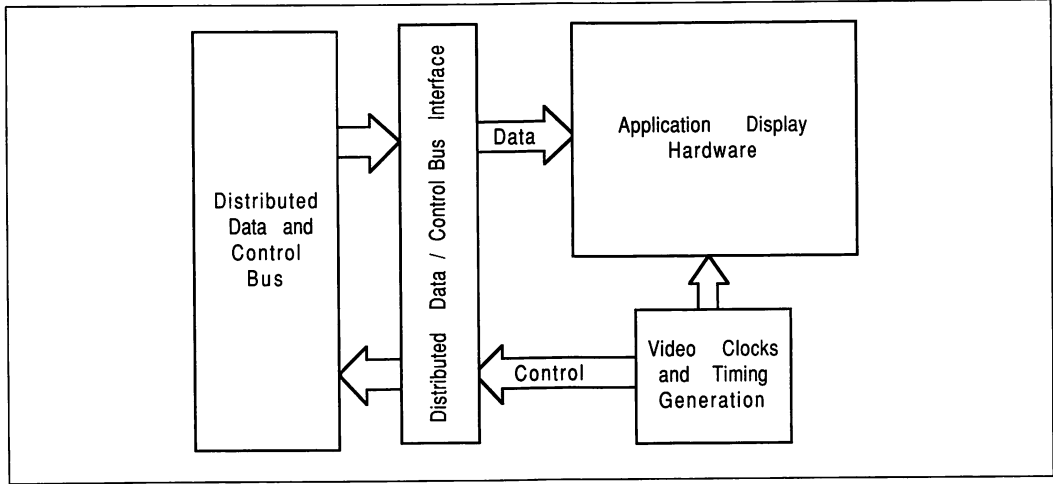
Figure 12.15 Generic display TRAMs

The Display Backend TRAM consists of:

• **A transputer link:** Communication to this module via at least one INMOS link, as a processor may not be necessary as it is used only for control and initialisation of the backend hardware.

• **Video system clock generator:** This provides the video system clock. The video system is timed from this clock.

• **A video timing generator:** From this, all synchronisation and system control is derived.

• **Serial control and data bus interface:** This drives the distributed serial control bus and takes data from the distributed data bus.

• **Application specific display hardware:** This hardware produces the application specific output derived from the 32 bit input data.

12.4   Serial port TRAM

In the short introduction to the serial port TRAM (section 12.3.5 and in figure 12.14) the functional blocks were briefly discussed. This section will discuss the serial port TRAM in more detail.

12.4.1   Introduction

The serial port TRAM can be considered as a transputer with memory, some of which is dual ported video RAM. The VRAM has a serial and a random access port to the frame store. These two ports can be considered more or less as separate entities ,(see figure 12.14). This section will give an overview of the serial port TRAM and then describe each port separately.

Memory map
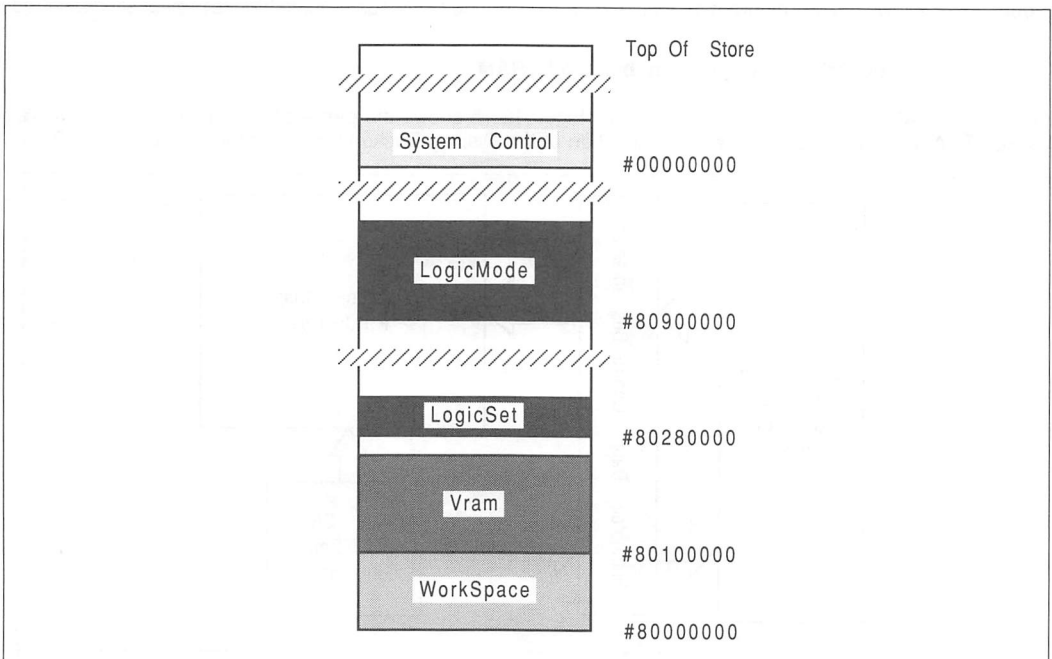
Figure 12.16 Memory map

The serial port module has 2.25 Mbytes of usable dynamic RAM. Of this 1 MByte is conventional dynamic RAM and 1.25 Mbytes is dual ported video RAM. Referring to figure 12.16, the RAM has been placed so that the video RAM abuts the 1 Mbyte of workspace RAM, this allows the video RAM to be used as extra workspace RAM if required.

The video RAM is mapped twice into the decoded memory map so that the special logic modes (marked **Logic Mode**) used in some video RAMs, which need special interfacing cycling, can be used (see section 12.9). These special logic modes can be set by writing data to the area of store reserved for this purpose (marked **Logic Set**). Registers which control the serial port addressing and frame synchronisation are mapped into the positive address space (marked **System Control**).

Frame store addressing and the video RAM

The serial port TRAMs frame store is designed around the **Packed Pixel** architecture (see section 12.2.3). There are two addressing schemes that can be used with video RAMs, when using packed pixel architecture:

• **Memory relative:** Data is placed into the frame store with addressing related to the physical addressing of the video RAM. Put simply, the VRAM row and column addresses have a direct relationship with the frame stores X and Y coordinates, but the display can have a different horizontal dimension than the frame store. Notice that the maximum width of display is the size of the dual port buffer in the VRAM, ie. 1024 8 bit pixels (see figure 12.17).
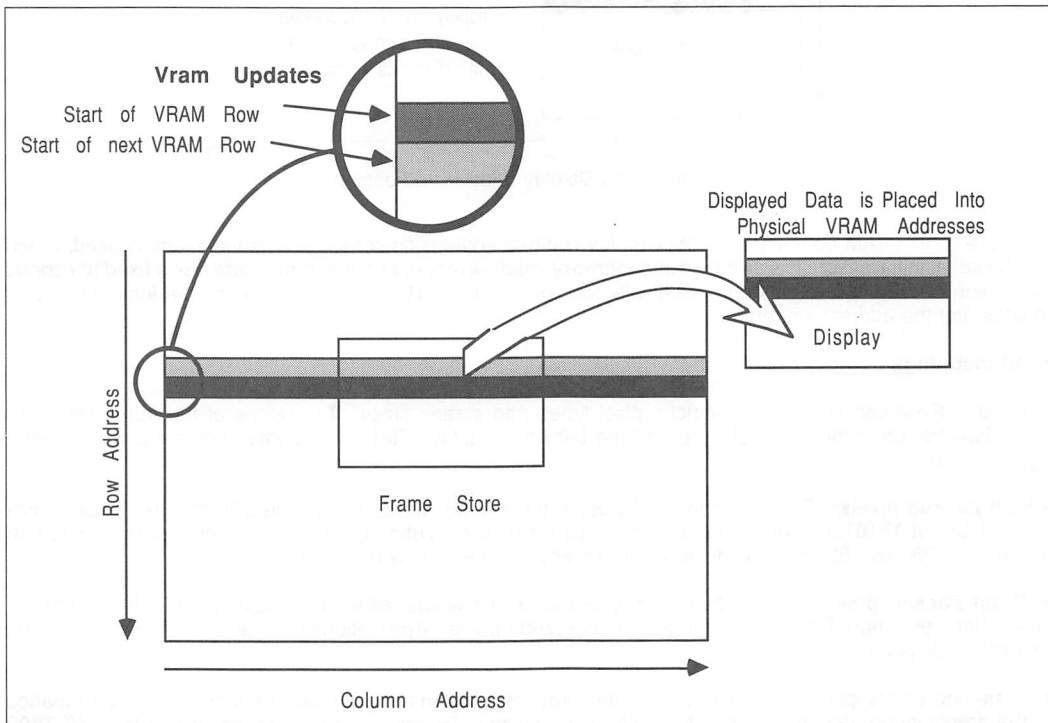
Figure 12.17 Frame buffer relative addressing

• **Display relative:** The VRAM row and column addressing have no direct relationship to the frame stores X and Y coordinates. Instead the frame store addressing and the visible display have the same horizontal dimension (see figure 12.18). This scheme needs the video RAM real time data transfer mechanism (see Section 12.9), which allows the display horizontal dimension to be longer than the VRAM dual port buffer, ie. longer than 1024 8 bit pixels.
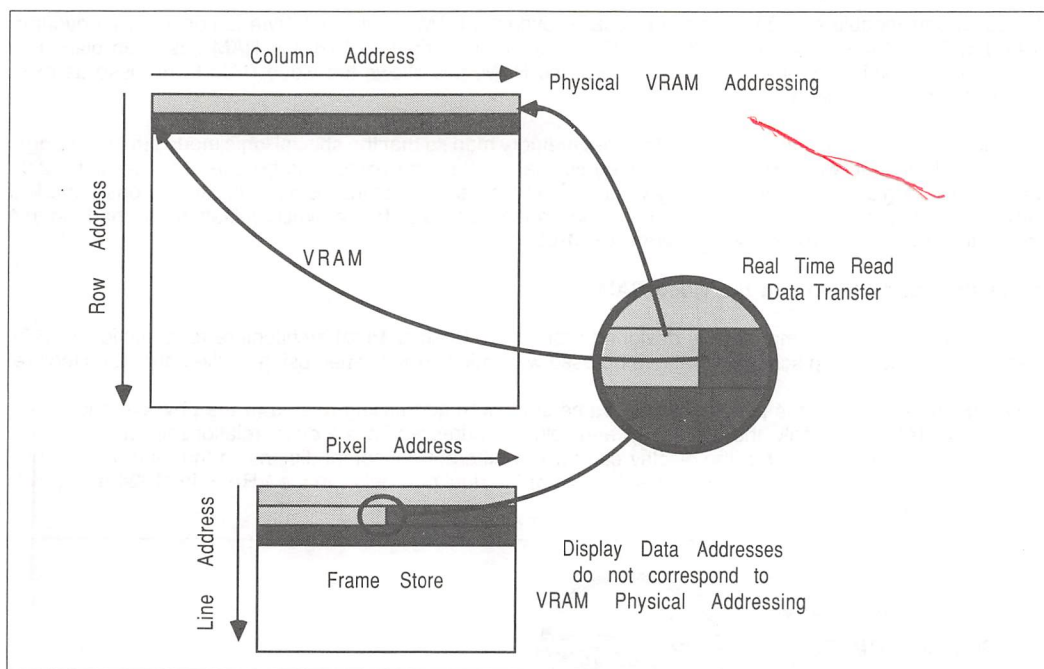
Figure 12.18 Display relative addressing

The serial port TRAM normally uses the **display relative** addressing scheme. When interlace is used, which can be set at initialisation, it is switched into **memory relative** mode, and the frame store has a fixed horizontal dimension of 1024 bytes (although the display can be smaller). These methods reduce the logic necessary to construct the address generator.

Pixel mappings

The video RAM can be used for various pixel types and screen sizes. The usage of the frame store entirely depends upon the user software and the backend display TRAM. Recommended mappings are (see figure 12.19):

• **8 bit packed pixels:** Pixels mapped as bytes, four pixels per word. This allows 256 colours per pixel with a maximum of 1310720 pixels. This can be used for high resolution CAD, ie. one serial port module can produce a 1280 by 1024 by 8 bit display, with an appropriate display backend.

• **32 bit packed pixels:** Pixels can be mapped as 32 bit words, allowing a maximum of 2^{32} colours per pixel. One serial port TRAM can have a total of 327680 pixels. Applications include any system that needs real colour displays.

The method of mapping the frame store to the processor can have a profound effect on the performance of the graphical operations a single IMS T800 can achieve. To achieve most efficient use of the IMS T800 performance, pixels should be mapped as either bytes or 32 bit word data types as this takes advantage of the IMS T800s internal datapath representation.

Double buffered frame store addressing

It is useful, when maximising performance in some graphic applications such as animation, to have at least two displays mapped onto the frame store. This allows one to be displayed whilst another is being updated.
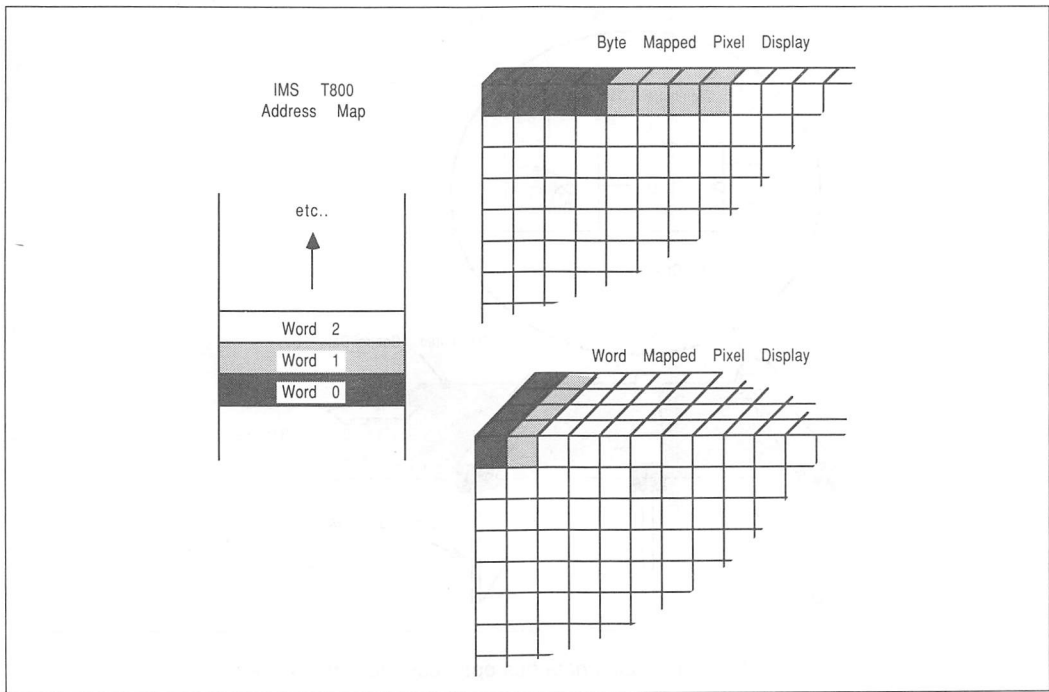
Figure 12.19 Pixel mapping

To facilitate this, the address of the first pixel at the top left of the display can be preset. This address presetting allows the display to be *flipped* to alternate areas of the frame store (see section 12.4.3). Flipping the display during frame flyback allows complete frames to be drawn before being displayed. This prevents disturbing visual artefacts.

The transputer can be informed of the state of the frame flyback condition so as to synchronise the frame flip to the frame flyback period. It is also sometimes necessary to synchronise with other serial port TRAMs in a system when some system wide or *global* event has occurred. Each serial port TRAM can cause a system event or can respond to it from an external source.

For this reason logic has been included so that the serial port TRAM can be informed when a frame flyback or system event has occurred. This logic uses the IMS T800 **Event** input (similar to a transputer link but it is only able to convey information about *when* external events have occurred). Alternatively the transputer can poll some registers which have bits representing the state of these signals.

Frame store distribution

The method of frame store distribution (see section 12.3.1) can have dramatic effects upon the design of the hardware to implement it. For the serial port TRAM the design rests on the specification of the **distributed data bus**, which consists of a synchronous (clocked) inverted open-collector bus. (see figure 12.20).

The open-collector arrangement allows any serial port TRAM to output data onto the bus at any time without fear of bus contention. This removes any need for any bus arbitration logic hence, allows arbitrary distribution of screen space amongst an arbitrary number of serial port TRAMs. Each serial port TRAM has enough memory to be able to address any pixel of the display. Since all serial port TRAMs are synchronised any one of them can alter the pixel data presently on the distributed data bus. If any serial port TRAM is not responsible for any particular pixel, it simply writes a null (zero) into that location in the frame store. This fits neatly into the IMS T800 2D block move instructions (see section 12.3.2), as null has special meaning when
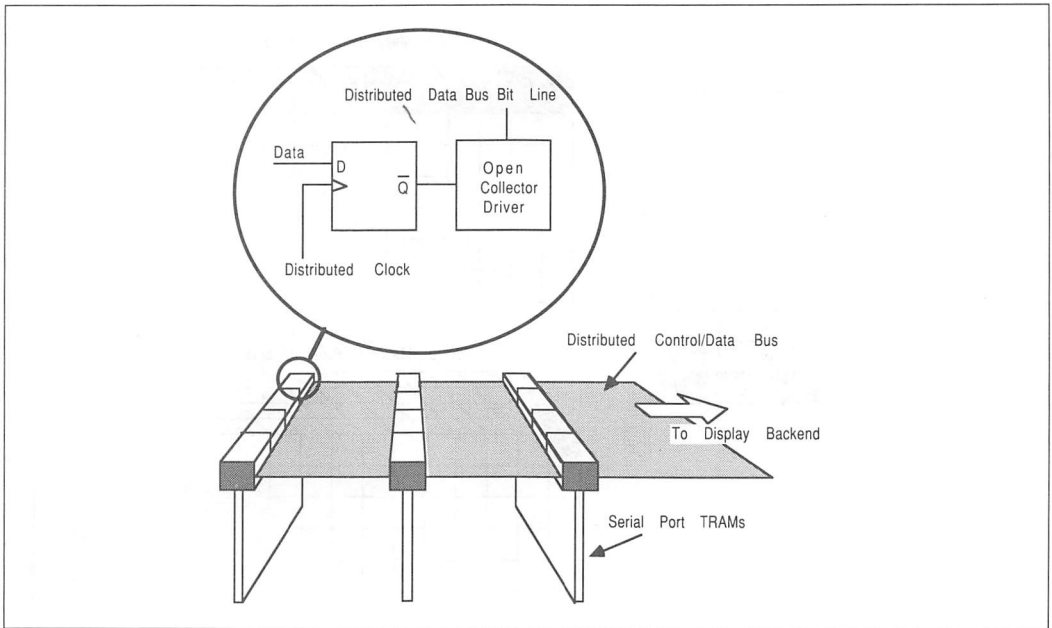
Figure 12.20 Distributed data bus open-collector arrangement

moving data with these instructions [2].

This distribution technique is simple, and provides the spatial and characteristic distribution methods described in section 12.3.1. To further enhance the flexibility of this, an output enable control bit is mapped into the IMS T800 address space. Any serial port TRAM output can be *switched off* (or nulled) completely. This provides the chronological distribution method discussed in section 12.3.1.

The objective distribution method also discussed in section 12.3.1 has not been implemented due to its complex nature. It is suggested that the reader refer to 6 and 7 both of which deal with distribution of solid object geometry and some implementation methods.

12.4.2   Random access port

This section will describe the implementation of the transputers access to the frame store. It also describes the mechanisms used to take full advantage of video RAM architecture.

Memory upgrades

As memory technology progresses, memory speeds increase as well as memory densities. Usually a designer, where possible, will incorporate the logic and PCB tracking necessary for a memory upgrade. To upgrade designs to more memory is quite straightforward, but to upgrade to a higher speed can mean a redesign, an option that can be economically unacceptable.

The IMS T800 allows the designer to upgrade memory speeds by changing the memory interface **Configuration** (see section 12.8.9). The serial port TRAM has the configuration data stored in a PAL (programmable array logic) which also controls the IMS T800s speed selection (as this has a bearing on the memory interface timings). This means that a speed upgrade requires only a PAL change (assuming logic delays are taken into consideration).

The upgrade paths allowed for in the design of the serial port TRAM are: • **Memory size:** An increase in

the size of the workspace RAM from 1 Mbyte to 4 Mbytes, using 4 Mbit rams when available. For the 4 Mbit RAMs extra addressing bits were included with no real cost. The upgrade involves a decode PAL and an option resistor (to change an address bit to a decoding PAL). The decoding needs to be changed because the video RAM will be pushed further up the address space.

- **Memory speed:** The speed of the interface can also be changed with the configuration PAL which also contains the speed selection for the IMS T800 as discussed above.

Memory cycles

The serial port TRAM has eight different types of memory access:

- **Internal read/write:** This cycle is the fastest. It is internal to the IMS T800 and lasts for a single cycle (50 nano seconds on the 20 Mhz transputers)

- **External read/write:** This cycle is the basic external memory cycle. It lasts for four processor cycles (200 nano seconds on the 20 Mhz transputer) and consists of a conventional dynamic RAM multiplexed addressed cycle (see figure 12.21).
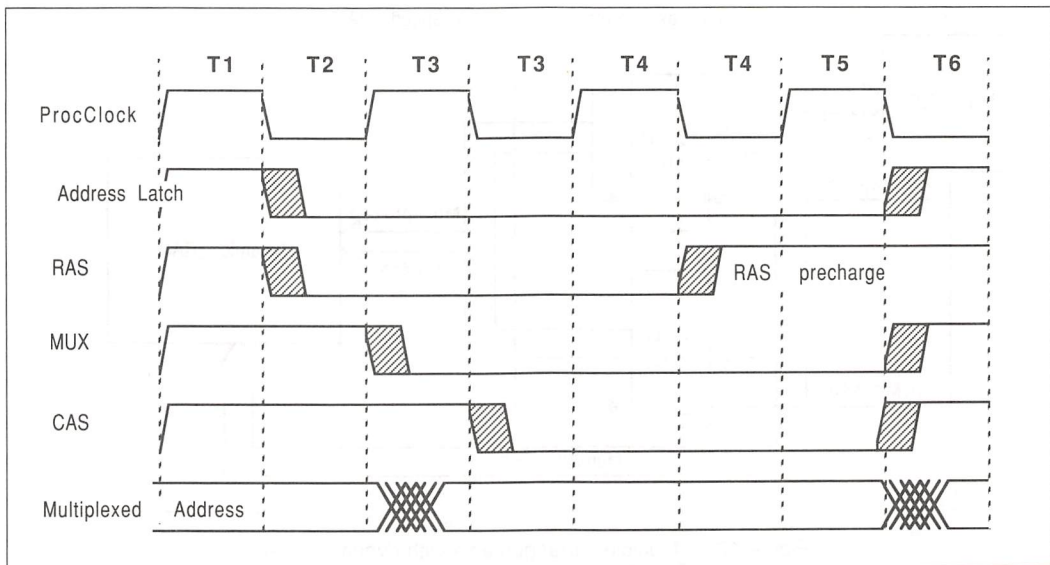
Figure 12.21 External read/write cycle

- **Refresh:** This is a CAS before RAS refresh cycle (see section 12.8.5), due to an addressing complication of the video RAMs . The **notMemRf** strobe is used to cause the relative timings of RAS and CAS to change.

- **Video update:** This cycle is controlled by the video update logic. It allows the video RAM serial port to be updated. The video logic proceeds after gaining control of the data and multiplexed address buses and cycles the video RAM with a serial port update cycle. This cycle only happens infrequently, when data in the serial port is about to run out of data.

- **Logic operation set:** The logic operation unit available in some video RAMs is activated using a CAS before RAS write cycle (see section 12.9). The logic mode remains set until a **Reset Logic Mode** or another **Logic Operation Set Mode** is issued.

- **Logic operation:** The **Logic Operation** cycle is a conventional RAS-CAS cycle but is six cycles long. This cycle needs a special extended RAS pulse, which is generated from a combination of the interface strobes **notMemS1**, **notMemS2** and **notMemS4**. This cycle is forced to six cycles using **notMemS4** strobe fed back

into the **Wait** input of the IMS T800. This is done as a function of the addressing, and is controlled by a PAL.

• **Serial port control logic:** This cycle allows the transputer to access the serial port control logic. It is initiated when **A31** is low. All RAMs are disabled in this cycle.

• **Configuration:** The configuration sequence is a conventional external read cycle that is used only after the transputer has just been reset (see section 12.8.9). The configuration data is generated from the configuration PAL using the six least significant unlatched address bits. The configuration data is then latched into a single bit of the decode address latch to hold the data until the end of the cycle.

Address latches and multiplexing

Due to the multiplexed address-data bus of the IMS T800 the addresses are only available at the begining of the external memory cycle. The addresses have to be demultiplexed from the data (see section 12.8.3). This is done using the transputer strobe **notMemS0** driving the latch enable inputs (marked **LE** on figure 12.22) of two ten bit transparent latches. The latches used are high speed CMOS, as these have low propagation delays and have high output drive.
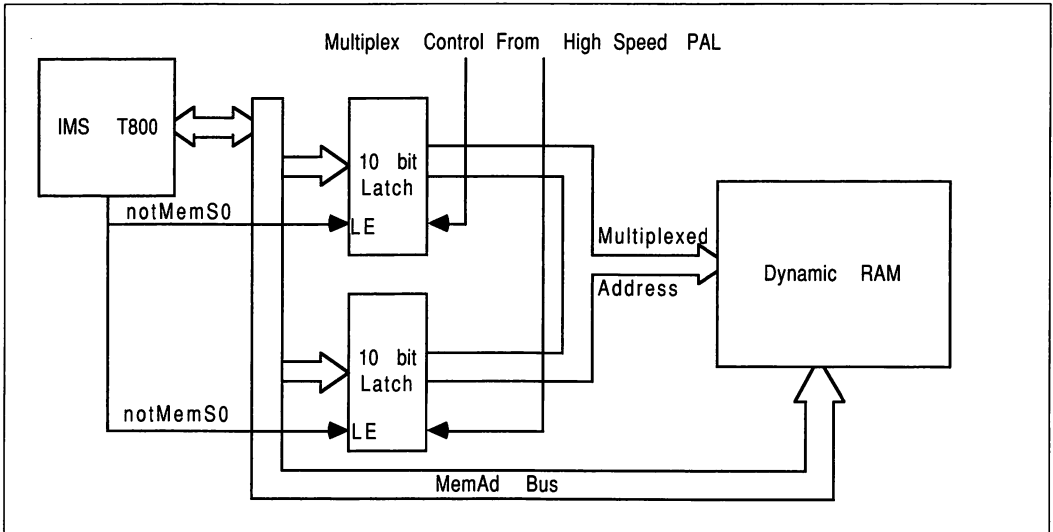
Figure 12.22 Multiplex arrangements with dynamic RAMs

Due to the multiplexed address bus used with dynamic RAMs, the now demultiplexed transputer addresses have to be multiplexed onto the RAM address bus (see figure 12.22). To achieve this the output enables of the address latches are controlled from a high speed PAL. The outputs from two latches are connected together.

This control is a function of the transputer memory interface strobes **notMemS2** and **MemGranted** (see Section 12.8). **MemGranted** is used because the video logic needs to drive the multiplexed address bus during a video update and therefore the multiplexer outputs have to be turned off completely.

A slight complication concerning the order of the multiplexed addresses presented to the video RAM, arises due to the way data is stored in the video ram. The most significant address bits are presented as row addresses, which can cause the a problem with the refresh address, which is on the low order address bits (see Memory cycles).

Decoding

The top address bits **AD31**, **AD23..18** and the **Configuration** data are latched into a separate eight bit transparent latch. These address bits are used for the decoding.

The RAM is arranged as:

- A single bank of general workspace RAM arranged as eight 256 Kbit by 4 RAMs (1 Mbit by 4 with the upgrade).

- Five banks of eight 64 Kbit by 4 (256 Kbit) video RAMs.

The high speed PAL that controls the operation of the address multiplexer also generates four RAS strobes, one for the workspace RAM and three for the video RAM. Pairs of video RAM banks share RAS strobes The last VRAM bank and the workspace RAM have their own RAS strobe.

The CAS strobes are supplied from another high speed PAL. This essentially is the RAM decoder, having six separate CAS strobes The decoding is a function of the latched addresses **A20..18**, **A31** and the **Option** input (see Memory upgrades). The **CAS** strobes are timed from **notMemS3** on a **External Read/Write** cycle.

Decoding with RAS is not essential if a full decode with CAS is used, as in this case, but it has several advantages:

- **Less heat dissipation:** It will cause less heat to generated by the memories. This is so because RAMs consume more current when RAS is cycled, even when not completely selected by a subsequent CAS strobe. Heat dissipation can be a problem in non forced air enclosures.

- **Speed:** Using several RAS strobes instead of one decreases the capacitive loading on the respective strobe, so the strobe can meet critical timings.

12.4.3   Serial access port

This section will describe the implementation of the serial interface on the serial port TRAM.

Introduction

At the heart of the distributed frame store are two clocks which are synchronous. Both clocks are distributed to all serial port TRAMs in the system. One is known as the **sequencer clock** and the other is known as the **VRAM clock** (the VRAM clock can run slower than the pixel rate, so that the 32 bits of data can be multiplexed at a higher clock rate to the display). The VRAM clock is stoppable, controlled by the display TRAM, and is switched off just before the start of, and switched on just before the end of, the horizontal blanking period.
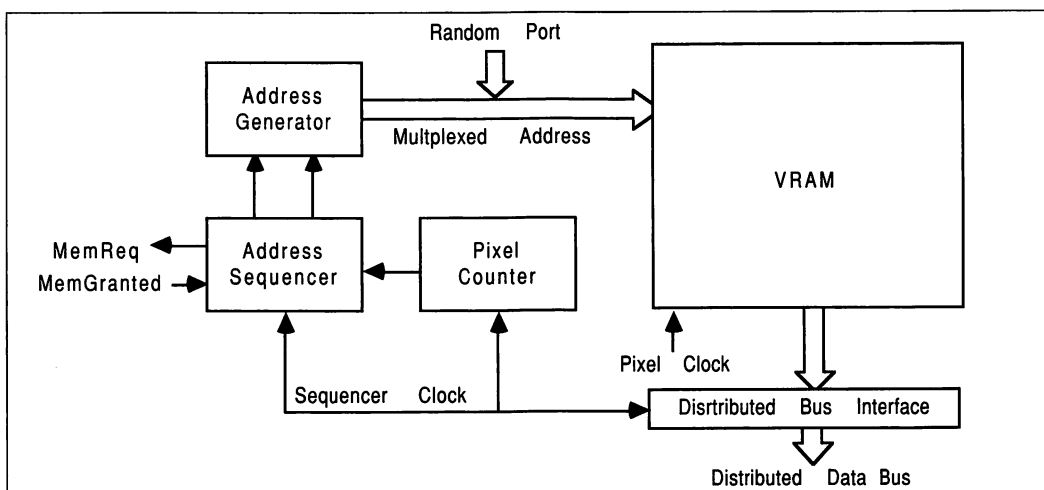
Figure 12.23 Serial interface block diagram

The serial port is built from several distinct groups of logic all synchronised to the previously mentioned clocks:

• **The address generator:** This generates the new serial address for the VRAM during a serial port update. The address generator has tri-state bus drivers connected to the multiplexed address bus of the VRAM.

• **Address sequencer:** This orchestrates control of the address generator during the update the serial port. The address sequencer takes over from the transputers memory interface and then cycles the VRAM in a data transfer cycle.

• **Pixel counter:** This starts the sequencer when serial data in the VRAM is about to run out. It is simply a counter that counts the data read out from the serial port, which resets itself immediately after the update occurs.

• **Serial bus interface:** This is the interface to the distributed data and control bus. This interface is clocked using the sequencer clock.

Address generator

The **address generator** is used when a video update cycle has been initiated. It provides 19 address bits, some of which are presented to the VRAM during a serial port update cycle (see section 12.9) and some of which are used as decode selectors. These addresses only form the **start address** for the serial data, subsequent data is accessed by clocking the VRAM (see figure 12.24).
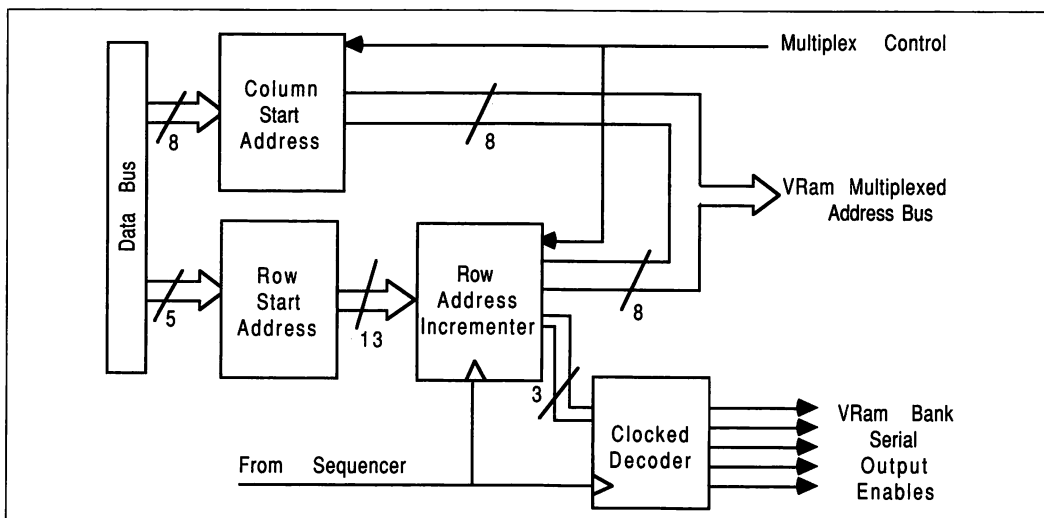
Figure 12.24 Address generation scheme

The lower 8 bits of the address are fixed but are presetable. This forms the column address to the VRAM during the update cycle. This determines which data appears at the VRAM serial output after the VRAM has been updated.

The next 11 address bits are generated from a preloadable counter that increments just after every update cycle. This address points to the first VRAM row to be accessed after each new frame is started. The lower 8 bits from this form the row address in the VRAM during the update cycle. The top 3 bits of the counter are used to control the serial output enables of the five banks of VRAM, see figure 12.24. There is no decoding on the update cycle, ie all VRAMs are updated at the same time.

The counters top 5 bits are preloaded from a 5 bit register which the user can preset so that the display can start from various addresses of the video ram. This provides the frame flipping mechanism mentioned in section 12.4.1.

Address sequencer

This logic interfaces the address generator to the VRAM and determines the timing of the serial update control strobes. It arbitrates this update cycle between the address generator and the IMS T800s memory interface logic.

The sequencer is designed to update the serial port without interrupting the pixel stream. To do this the pixel counter informs the sequencer that the serial data is about to run out. The entire sequencer operation last for 31 sequencer clocks, (new data appears at the VRAM serial outputs after 30 sequencer clock periods).

The sequencer requests the VRAM address bus by asserting **MemReq** (see section 12.8). When **Mem-Granted** is asserted by the transputer, the sequencer cycles the VRAM in a serial port update cycle. This cycle updates the serial port via the random port when the VRAM strobe **DT/OE** is brought high synchronised with the VRAM serial clock (see section 12.9). This is known as *Real Time Read Data Transfer*, see figure 12.36.

Pixel counter

The serial port of the VRAM wraps around after 256 clocks. It therefore needs reloading every 256 VRAM clock cycles if data is not to be redisplayed. To implement this, the pixel counter signals to the sequencer when the end of serial data is about to occur. This end of data signal knows that the update will occur 30 clock periods later, so it signals the sequencer early.

A slight complication of the sequencer operation concerns the line flyback period. The sequencer must finish its operation before line flyback occurs, otherwise data destined for the start of the next line will be lost. The pixel counter will not cause an update to occur if an end of line is due, so that the update cannot occur during the line flyback period. The timing of this is critical, as the data which finds its way to the display is pipelined twice (at the distributed data bus output driver and at the display TRAM) before getting to the display. This means the pipeline must be precharged with data before the display line starts and emptied before the line ends. To this end, the VRAM clock is turned on two clock periods before the start of the line and switched off two clocks before the end of the line.

Distributed control

The serial port TRAM is designed to function as part of a distributed graphics system. For this reason the control necessary to drive the distributed data bus has to be common to all serial port TRAMs in the system. All clocking and control strobes are distributed using parallel terminated transmission lines.

The transmission lines are driven at the source (Display TRAM) using high speed CMOS logic with high output drive capability. This is terminated with a resistor to ground equal to the characteristic impedance of the transmission cable (this resistance will be anything between 50 and 100 Ω). All control inputs to the serial port TRAM are short stubs to buffers, which offers little disturbance to the transmission line.

12.5   Display TRAMs

12.5.1   Introduction

It would be impractical to build a graphics system that is capable of practically any present day graphical display output. It is reasonable that a display TRAM should have application specific display output driving hardware.

12.5.2   An example display TRAM

This particular display TRAM has been designed with some features that allow it to be used in a variety of applications. This display TRAM has:

• **A transputer:** A **IMS T212** is used purely as a logic controller to initialise the video timing logic, colour look up tables and the mode selection.

• **Distributed control bus interface:** This consists of a few transmission line drivers, distributing the control signals to all the serial port TRAMs.

• **Video clocks and timing generator:** The pixel clocks and video timing generation used to synchronise all serial port TRAMs are controlled by the display TRAM.

• **Three pixel channels:** Each display channel converts 32 bits of input data from three distributed data bus inputs into the analogue control signals to drive standard display monitors.

Pixel channels

The display TRAM consists of three independent 8 bit pixel channels, all with common clock and video timing generators (see figure 12.25). Each channel has:

• **Premultiplexer:** A eight bit premultiplexer which links 8 bits of data from channel 0 onto channel 1 and 8 bits of data from channel 0 onto channel 2. This then maps 24 input bits of channel 0 onto the lowest 8 bits of channels 0,1 and 2.

• **Input latch:** Distributed data bus 32 bit input latch.

• **Multiplexer:** 32 bit input 4 to 1 multiplexer

• **CLUT:** 256 location colour lookup table.
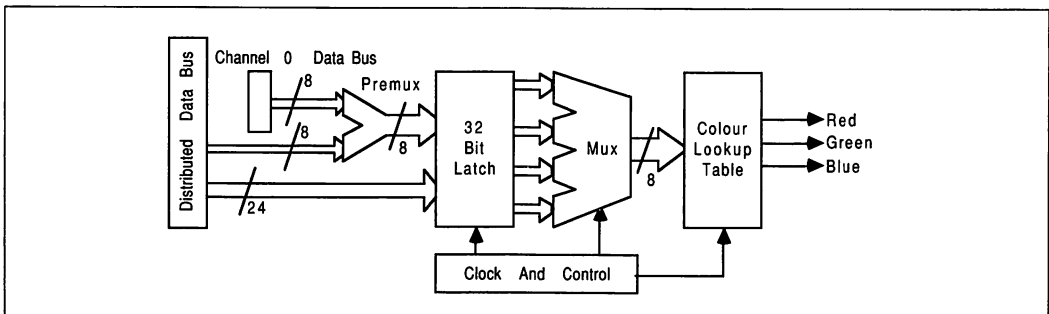
Figure 12.25 Pixel channels

Display modes

There are three modes that the display TRAM has been designed for:

• **8 bit mode:** This mode treats the 32 bit pixel data entering the display TRAM as four 8 bit pixel values. This data is multiplexed to the colour look up table. All three pixel channels operate separately sharing only the distributed control, (see figure 12.26).
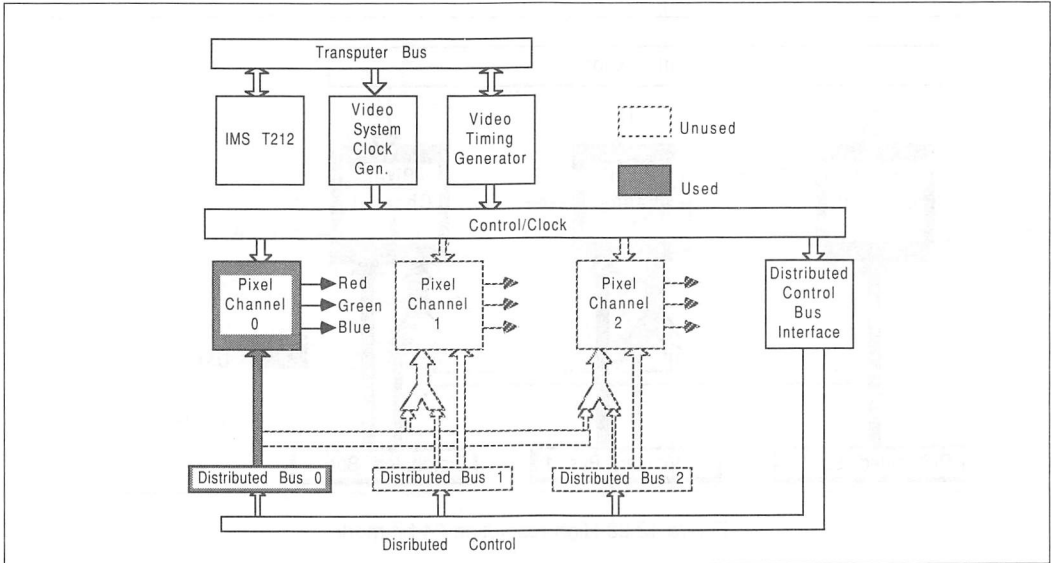
Figure 12.26 8 bit mode

• **Low resolution 24 bit mode:** This mode treats the 32 bit pixel data entering the display TRAM as a single 32 bit word of pixel data. The top 8 bits are not used, leaving the lower 24 bits as pixel data. The three pixel channels contribute to the display, one channel per primary colour (see figure 12.27).
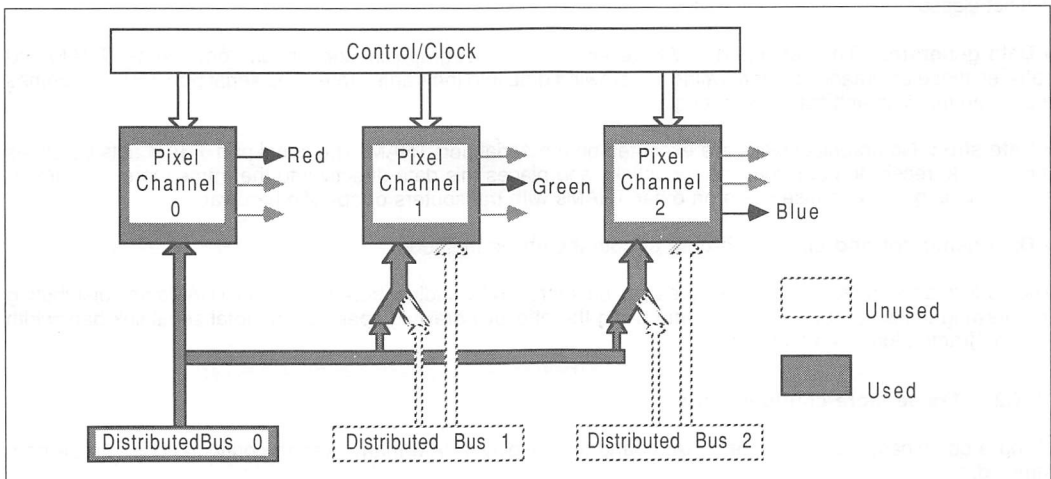
Figure 12.27 24 bit mode

The 24 bit mode has a different clocking arrangement. Since data is being displayed at the same clock speed (pixel clock) but four times as much data is being used by the display, the input clock speed must be increased, ie pixel clock runs at the same speed as the pixel bus. The mode selection can change the clocking arrangements to suit these modes.

• **High resolution 24 bit mode:** This mode is similar to the 8 bit mode, except all three channels are used to provide each of the primary colours (see figure 12.28).
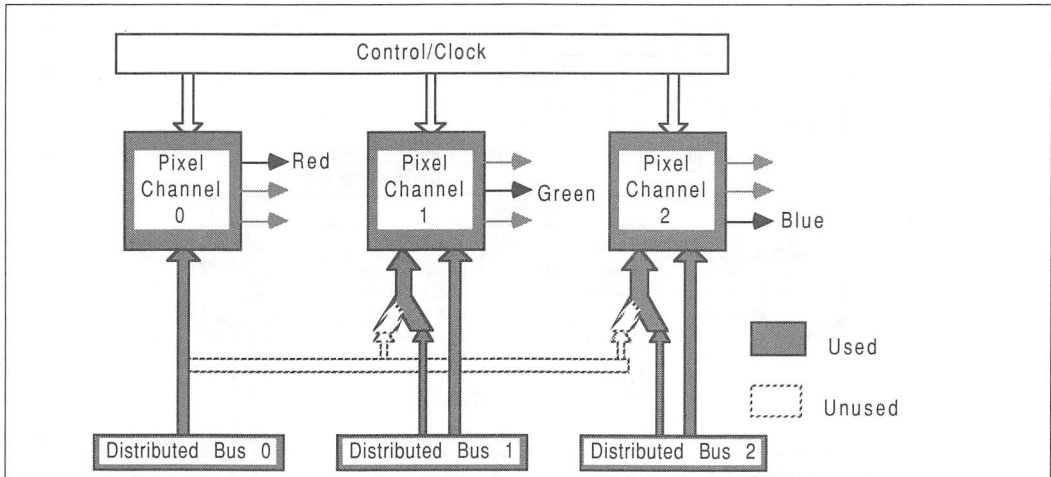
Figure 12.28 High resolution 24 bit mode

12.6   System configurations

12.6.1   Driving the frame store

The serial port TRAM can be used in a varied and non specific manner, but the techniques fall into several distinct classes.

• **Data generator:** The serial port TRAM receives high level graphical commands from another TRAM and satisfies these commands by generating the drawing data into the frame store. The serial port TRAM becomes a programmable graphical drawing engine.

• **Data sink:** No graphical tasks are executed on the serial port TRAM. The serial port TRAM acts purely as a data sink, receiving data from the serial links and places this data directly into the frame store. The frame store data is generated elsewhere on other TRAMs with transputers or specific hardware.

• **Data generator and sink:** A mixture of both the above methods.

The performance of the above techniques can be improved by adding more Serial Port TRAMs and distributing the drawing tasks appropriately, thus improving the effective drawing speed or the total serial link bandwidth into the frame store (see figure 12.29).

12.6.2   Frame store configurations

Using a combination of serial port TRAMs and the Display TRAM many system configurations can be constructed.

• **Minimal 8 bit display system:** The minimal system consists of a single serial port TRAM and is connected as shown in figure 12.13. This minimal system provides all that is necessary for a 8 bit pixel (256 colour)
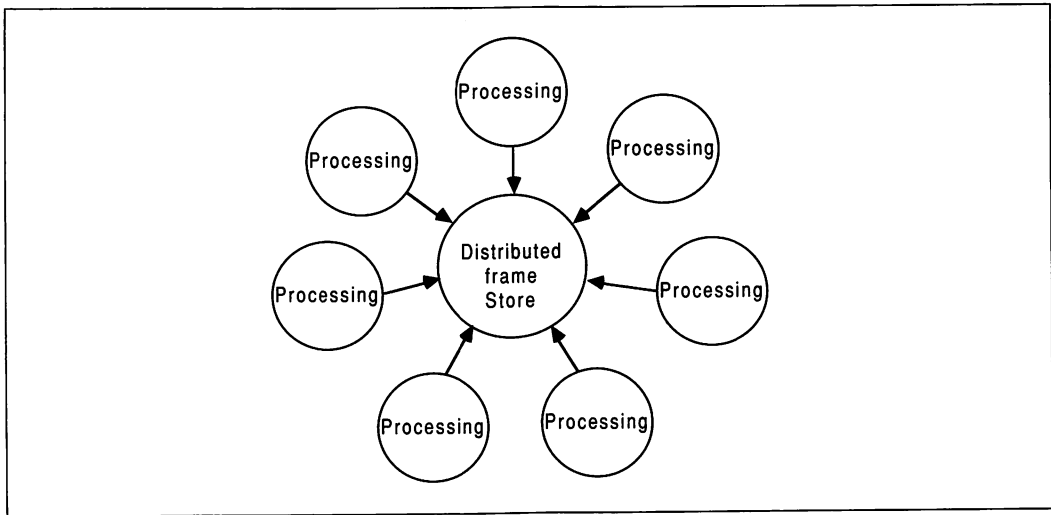
Figure 12.29 Conceptualisation of the distributed frame Store

graphic display, to a maximum of 1280 by 1024 pixels.

● **Distributed 8 bit display system:** Figure 12.13 shows a distributed 8 bit graphic display system. This distribution provides increased drawing speed and transputer link bandwidth into the frame store.

For example in [7], a multi-user flight simulator is described in detail. The system produces an 8 bit 512 by 512 pixel display at 23 frames/sec. The system is based upon a transformation pipeline, and at the end of the pipeline are the polygon shaders. These are transputers that produce display data and send it to the graphics transputer using the data sink method described in section 12.6.1. An upgrade to higher resolution would consist of placing these polygon shaders onto four serial port TRAMs, turning the display system into a data generator (see figure 12.30). The display resolution can now be increased with no impact on performance.
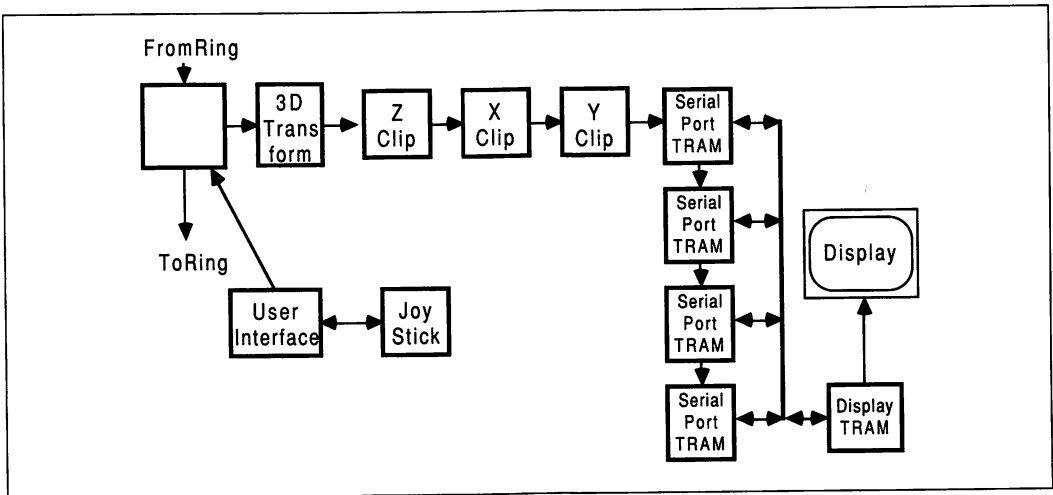
Figure 12.30 Modified high resolution flight simulator

• **Minimal low resolution 24 bit display system:** The system in figure 12.13 can also be used as a low resolution (maximum of 327680 pixels) 32 bit pixel system. The Display TRAMs premultiplexer is used in this configuration and provides a maximum of 24 bits of output colour (8 bits per primary). Each pixel channel is used as a single primary colour output.

• **Distributed low resolution 24 bit display system:** The system in figure 12.13 can also provide a low resolution 32 bit display. The display TRAM is set into 24 bit mode as above, but the system provides increased possible drawing and link bandwidth into the frame store as in the distributed 8 bit system, but with more colours.

• **High resolution 24 bit display system:** This system (figure 12.31) is essentially 3 separate 8 bit systems. This method separates the red green and blue components into three 8 bit high resolution display channels as in the 8 bit system. It has all the characteristics of the 8 bit system but each of the 3 pixel channels on the Display TRAM operate independently to provide a primary colour as in the low resolution 24 bit system.

• **High resolution distributed 24 bit display system:** This system (figure 12.31) is essentially the same as the previous system except that each 8 bit pixel channel is distributed in the same way as the 8 bit system. Again this method separates the red green and blue components into three 8 bit high resolution display channels, but the possible drawing and link bandwidth into the frame store has been increased.
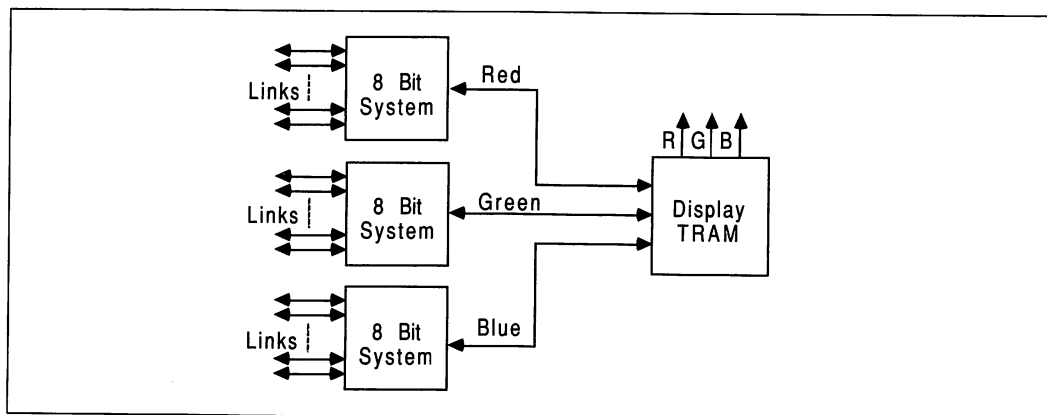
Figure 12.31 High resolution 24 bit display

12.7   Conclusion

This technical note has shown that the performance of the frame store can be increased without using special hardware by using video RAMs. The video RAM provides a flexible and efficient frame store by mapping the display data directly onto the transputers address map without degradation of bus usage.

This note has looked at the problems associated with frame stores, and has highlighted the problems of single processor bus bottlenecks. It has shown how these bottlenecks can be removed by distributing the frame store, and that this distribution is simplified using transputers.

It has been shown that the large amount of processing necessary to perform typical graphical operations rapidly swamps single processor systems. In high performance systems it becomes necessary to distribute the processing task into smaller more manageable tasks. The complexity and control of this distribution is considerably reduced using transputers and occam, and the distribution of the frame store compliments such a system by providing a convenient interface to the display. Once the distribution has been achieved, adding more transputers into the system, at the display or at the processing front end, can produce any desired system performance.

12.8   Transputer memory interface

The IMS T800 has a configurable memory interface designed to allow easy interfacing of a variety of memory types with a minimum of extra components. The interface can directly support DRAMs, SRAMs, ROMs and memory mapped peripherals.

The IMS T800 has a 32 bit multiplexed data and address bus with a linear address space of 4 Gbytes. The interface has:

- 4 byte write strobes, for controlling byte write operations.

- A read strobe.

- A refresh strobe, for signalling refresh cycles when using dynamic RAMs.

- 5 configurable strobes, for general interfacing of memories.

- A wait input, for extending the interface period.

- A memory configuration input, used to configure the interface at after reset.

- A bus request input and bus grant output, to relinquish control of the memory interface.

Figure 12.32 shows the inputs and outputs for the T800 transputer that are associated with the memory interface.

notMemWrB0-3	——4——▶	byte write strobes
notMemRd	——————▶	read strobe
notMemRf	——————▶	refresh strobe
notMemS0-4	——5——▶	configurable strobes
MemnotWrD0	◀—————▶	notWriteFlag/data 0
MemnotRfD1	◀—————▶	notRefreshFlag/data 1
MemAD2-31	◀—30——▶	address/data 2-31
MemReq	◀—————	external request
MemGranted	—————▶	external request granted
MemWait	◀—————	wait states
MemConfig	◀—————	configuration input
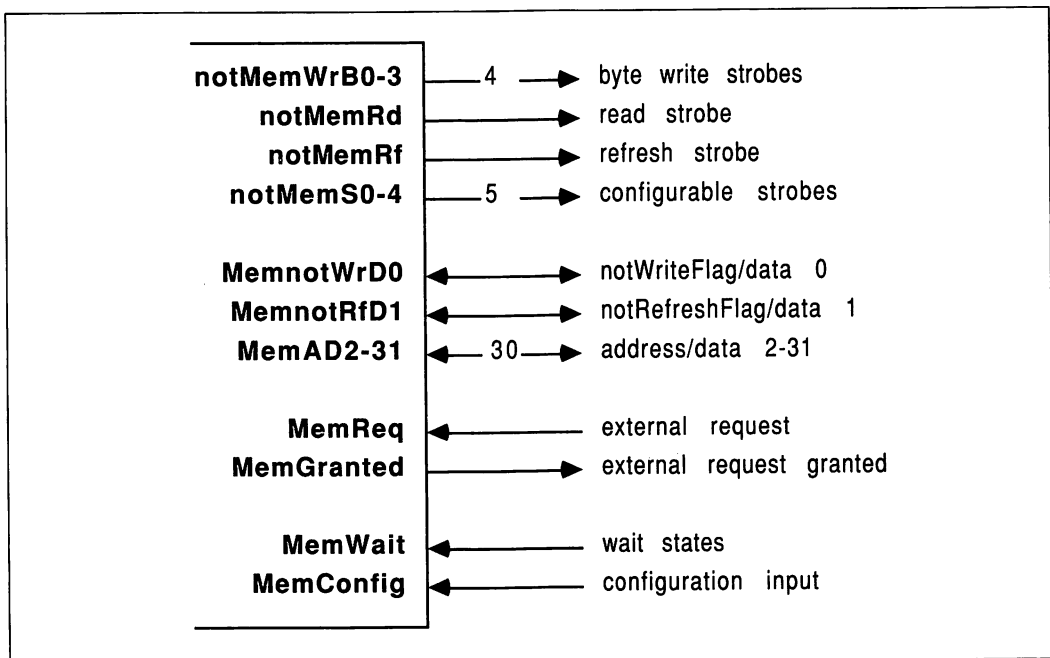
Figure 12.32 IMS T800 memory interface

All RAM appears to the IMS T800 as 2^{32} bytes mapped as 32 bit words in a linear signed address space. Addresses, therefore, run from 80000000_{16} through $FFFFFFFF_{16}$ to $7FFFFFFF_{16}$. As shown in figure 12.33 the IMS T800 has 4 Kbytes of internal single cycle (50ns on 20 Mhz part) RAM from byte address 80000000_{16} to $80000FFF_{16}$. Of this RAM the first 70_{16} bytes are reserved for processor use. The IMS T800 has MemStart at 80000070_{16} and start of external memory at 80001000_{16}.

Machine   Map

| Reset   Inst | | #7FFFFFFE |

| Memory   Configuation | #7FFFFFF8 |
| | #7FFFFF6C |

User   Programs

#80001000   External   Mem
#80000070   Mem   Start

Reseverd   For

Extended   Functions

#80000000 Base Of Memory

Hi   Lo

Figure 12.33 T800 memory map

It is advisable for the address range 80000000_{16} to $FFFFFFFF_{16}$ to be used for RAM and 00000000_{16} to $7FFFFFFF_{16}$ to be used for ROM and I/O. If external memory exists it will overlap internal memory, but if the memory map is not completely decoded, it is usually possible to access the *hidden* external memory at another address.

12.8.1   Memory interface timing

The IMS T800 memory interface cycle has six timing states, referred to as **Tstates**. The **Tstates** have the nominal functions:

Tstate

T1	address setup time before address valid strobe
T2	address hold time after address valid strobe
T3	read cycle tristate/write cycle data setup
T4	extended for wait states
T5	read or write data
T6	end tristate/data hold

The duration of each **Tstate** is configurable to suit the memory devices used and can be from one to four **Tm** periods. One **Tm** period is half the processor cycle time, i.e. half the period of **ProcClockOut**. Thus, **Tm** is 25 nsec for an IMS T800-20 (20MHz transputer). **T4** may be extended by wait states in the form of additional **Tm**s.

With this flexible arrangement, a variety of memory timing controls can be obtained with little external hardware. The bus timing is shown in figure 12.34.

Every memory interface cycle must consist of a number of complete cycles of **ProcClockOut**: i.e. it must consist of an even number of **Tm**s. If there are an odd number of **Tm** periods up to and including **T6**, an extra **Tm** shown as "E" by the memory interface program (see section 12.8.9) will be inserted after **T6**.

12.8.2   Configurable strobes

The use of the strobes **notMemS0** to **notMemS4** will depend upon the memory system. The rising edge of **notMemS1** and the falling edges of **notMemS2** to **notMemS4** can be configured to occur from 1 to 31 **Tm** periods after the start of **T2**. This is summarised in figure 12.34 and in the table below.

Figure 12.34 The configurable strobes

Signal	Starts	Ends
notMemS0	T2	T6
notMemS1	T2	T2 + (Tm*s1) (or end of T6 if this occurs first)
notMemS2	T2 + (Tm*s2)	T6
notMemS3	T2 + (Tm*s3)	T6
notMemS4	T2 + (Tm*s4)	T6

Where **s1**, **s2**, **s3** and **s4** are the configured number of **Tm** periods for each respective strobe.

It should be noted that the use of wait states can advance the rising edge of **notMemS1** in relation to that of the other strobes. Care must be taken if this signal is being used when **Wait** states are being used.

12.8.3   Multiplexed address-data bus

The address and data buses are multiplexed onto the **MemAd** bus. Addresses are available from the begining of the cycle until the end of **T2**. Whereupon the **MemAd** bus will go either tri-state (a passive state) or have data present depending whether a read or write cycle is in progress (if the cycle is a single or multiple byte-write cycle, bytes which are not to be written will go tri-state)

The address bus can be demultiplexed using transparent latches (latches that act as buffers until the latch control is used, whereupon the data becomes held), controlled by **notMemS0** directly (not a configurable strobe). The transparent latch will buffer the **MemAd** bus whilst **notMemS0** is not active. When **notMemS0** goes active at start of **T2**, the addresses are held. Using transparent latches makes the demultiplexing simple (using **notMemS0** directly) and gives as much address set up time as possible.

12.8.4   Byte selection

During a write cycle, byte addressing is achieved by the four write byte strobes **notMemWrB[0..3]**. Only the write strobes corresponding to the bytes to be written are active. During a read cycle complete words are read, and the bytes to be used are selected internally. Thus, the two lowest order address bits **A0** and **A1** are not needed and are not output with the rest of the addresses. However, care must be taken when mapping

byte wide peripherals onto the interface, as they are addressed on word boundaries.

The two lowest order data bits during the address period, are used to give early indication of the type of cycle which is in progress:

>   **MemnotWrD0** is low during **T1** and **T2** of a write cycle.

>   **MemnotRfD1** is low during **T1** and **T2** of a refresh cycle.

The **notMemWrB** strobes can be configured to fall either at the beginning of **T3** (early write) or at the beginning of **T4** (late write); the rising edge is always at the beginning of **T6**. Early write gives a longer set up time for the write strobe but data is only valid on the rising edge of the pulse. For late write, data is valid on the falling edge of the strobe but the pulse is shorter.

12.8.5   Refresh

The IMS T800 has an on-chip refresh controller and 10 bit refresh address counter and can, therefore, refresh DRAMs of up to 4 Mbit capacity (since these are arranged as 1024 rows of 4096 bit columns) without requiring the counter to be extended externally.

Refresh can be configured to be either enabled or disabled. If enabled, the refresh interval can be configured to be 18, 36, 54 or 72 **ClockIn** periods; though if a refresh cycle is due, the current memory cycle is always completed first. The time between refresh cycles is thus almost independent of transputer speed and the length of memory cycles.

Refresh cycles are flagged by **notMemRf** going low before **T1** and remaining low until the end of **T6**. Refresh is also indicated by **MemnotRfD1** going low during **T1** and **T2** with the same timing as address signals. The address output during refresh is:

AD0	= **MemnotWrD0**	high, indicates a read
AD1	= **MemnotRfD1**	low, to indicate refresh
AD2 - AD11		refresh address
AD12 - AD30		high
AD31		low

During refresh cycles, the strobes **notMemS0** - **notMemS4** are generated as normal.

Several choices for the designer exist for refresh schemes with the **IMS T800**. These are :

RAS only Refresh
>   This requires an address supplied by the interface to refresh the selected row. The row address is incremented after every refresh cycle. Note that no **CAS** is necessary during refresh and all RAMs are **RAS** selected.

CAS Before RAS Refresh
>   This causes an internal counter in the RAM to be used as the refresh address. It requires that the **CAS** strobe goes active before the **RAS** strobe. This can be arranged because the **notMemRf** strobe is active at the beginning of memory cycle and appears at the same time as addresses and can therefore be used to switch the timing of the **RAS** and **CAS** strobes.

Where:

CAS Refers to the Column Address Strobe input on the dynamic RAMs.

RAS Refers to the Row Address Strobe input on the dynamic RAMs.

As all RAMs need to be refreshed simultaneously, all RAMs are **RAS** selected. As RAMs will consume current when **RAS** goes active, this is usually the most power hungry cycle of a dynamic RAM interface.

Care has to be taken to ensure that the power supply is not left with a problem of supplying high current

surges at refresh, and thereby causing a power supply noise. This can be a particular problem if many transputers with lots of dynamic RAM are used with a common power supply. The refresh may well be nearly synchronous due to the common reset signal. This problem will be made worse if the transputers have a common input clock. The clocking may be near synchronous (albeit on different phases due to the phase locked clock multiplyer on each transputer).

It is suggested that large capacitors are used as near to the dynamic RAM as possible, as this will reduce the supply noise to acceptable levels.

12.8.6   Wait states

Memory cycles can be extended by wait states. **MemWait** is sampled close to the falling edge of **ProcClock-Out** prior to, but not at, the end of **T4**. If it is high, **T4** is extended by additional **Tms** (shown as 'W' by the memory interface program). Wait states are inserted for as long as **MemWait** is held high, **T5** proceeds when **MemWait** is low. Note that the internal logic of the memory interface ensures that, if wait states are inserted, **T5** always begins on a rising edge of **ProcClockOut**: so the number of wait states inserted will be either always odd or always even, depending on the memory configuration being used.

12.8.7   MemReq, MemGranted and direct memory access

Direct memory access (DMA) with the IMS T800 has been implemented in the following way.

MemReq can be asserted asynchronously (at any time) with respect to **ProcClockOut**, but to guarantee DMA, **MemReq** must be set up two periods **Tm** before end of **T6**. **MemReq** will be sampled at at the final **Tm** period of **T6** of a refresh or external memory cycle when **ProcClockOut** is low. If the IMS T800 is accessing internal RAM or is idle, **MemReq** is sampled during the low period of every **ProcClockOut** and internal memory accesses will not be affected by this DMA activity.

When **MemReq** has been sampled high, two **Tm** periods after **ProcClockOut**s next rising edge, the address bus is tristated and all strobes go inactive. One **Tm** period later **MemGranted** is set high to indicate a DMA cycle is in progress. After this **MemReq** is sampled at each low period of **ProcClockOut** and if found to be low **MemGranted** will be removed synchronously at the next falling edge of **ProcClockOut**.

A few points to note about DMA:

- If the DMA period lasts for more than one refresh interval the DMA hardware is responsible for refresh.

- Refresh has higher priority than DMA. So the worst case asynchronous DMA response time is two external memory interface cycle periods (one external cycle plus one refresh cycle) plus 3 **Tm** periods.

12.8.8   Termination

This is always worth a mention, as it is frequently overlooked. All buffered memory strobes and multiplexed addresses should be series terminated with 25 to 50 Ω. This prevents negative voltage spikes on address and control pins. It cannot be overstressed that negative spikes can cause random memory failures, especially on the higher density RAMs.

The unbuffered data bus need not be terminated as the transputers output drive pads have been designed to prevent the fast edges associated with negative excursions.

12.8.9   Configuration of the memory interface

A memory interface configuration is specified by a 36 bit word and is fixed at reset time. The IMS T800 has a selection of 13 pre-programmed configurations. If none of these is suitable, a different configuration can be selected by supplying the complement of the configuration word to the IMS T800s **MemConfig** input immediately following reset.

A pre-programmed configuration is selected by connecting **MemConfig** to **MemnotWrD0**, **MemnotRfD1**, **MemAD2-MemAD11** or **MemAD31**. Immediately after reset, the IMS T800 takes all of the data lines high and then, beginning with **MemnotWrD0**, they are taken low, at intervals of two **Clockin** periods, in sequence. This is the internal configuration scan.

If **MemConfig** is high at the start of this scan, an internal configuration is to be selected. The selection is accomplished by **MemConfig** going low when the IMS T800 pulls a particular data line low, the configuration associated with that data line is then used.

If, at the begining of the scan, **MemConfig** is sensed low before **MemnotWrD0** goes low, an external configuration is selected. To aid this when an external configuration is used the configuration data is expected to be inverted so that a single inverter between a MemAd pin and the **MemConfig** signals an external configuration from ROM.

After the scan, the IMS T800 performs 36 configuration read cycles from locations $7FFFFF6C_{16}$ to $7FFFFFF8_{16}$. If an internal configuration was selected these reads are ignored. If an external configuration has been selected, each of the configuration read cycles will latch one bit of the configuration data into the **MemConfig** input from an external source.

Using an internal configuration has the advantage of requiring no external components, only a connection from **MemConfig** to the appropriate data line.

However, selecting an external configuration can also be very economical in component use if the configuration data is stored in a PAL and this PAL is used for other purposes concerning the low order address bits.

If the transputer is booting from ROM, the ROM must occupy the top of the address space. One bit of the memory configuration data can be stored in each of the 36 addresses mentioned above and the only additional hardware required is an inverter connecting the appropriate data line (usually **MemnotWrD0**) to **MemConfig**. **MemConfig** is thus held low until **MemnotWrD0** goes low and is fed with the inverse of the configuration data during the 36 read cycles. Alternatively, the inverted configuration data can be generated from **A2-A7** by a PAL.

12.8.10   The memory interface program

The INMOS Transputer Development System includes an interactive program which assists in the task of memory interface design. The program produces timing diagrams and timing information so that the designer can see the effects of varying the length of each **Tstate** and the positions of the programmable strobe edges. Of course, the program cannot allow for external logic delays and loading effects as these are system dependent but it does assist greatly in preliminary design. (It has sometimes been considered an essential tool in designing the interface configuration data).

A foolproof method to produce the PAL equations for the **configuration** data is to modify the configuration data page generated by the memory interface configuration program.

12.9   Video RAMs

12.9.1   What is a video RAM

Recent developments in RAM design architecture has made available a cost effective dual ported Video RAM. The video RAM has a secondary set of output selector register sets (see figure 12.35) controlled by an external serial clock.

This extra selector is able to operate totally asynchronously to the normal selector register set. These two register sets are referred to as the access ports to the RAM bulk, the random access port and the serial access port. The serial access port accesses data in a sequential manner, which needs to be updated when data runs out using the special update cycle from the random port.

The random access port is similar to conventional dynamic RAMs except for the extra function of sequencing the OE (Output Enable) pin. This extra function is called a **Data Transfer**, hence the pin is renamed DT/OE.
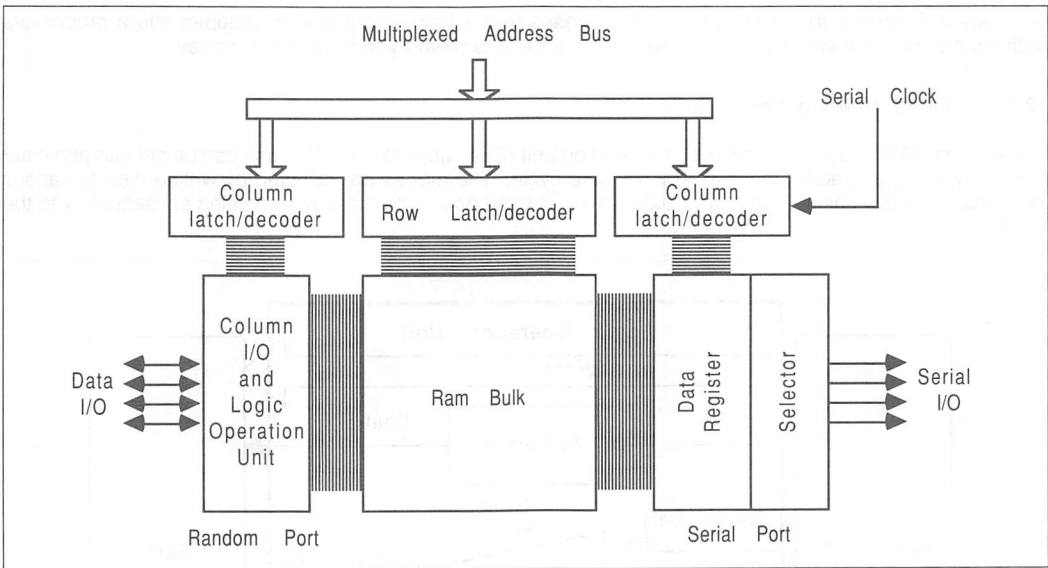
Figure 12.35 Video RAM architecture

Sequencing the DT/OE pin on a random access causes data transfer from the RAM to the serial port. Once the serial port is updated it can proceed to output data without recourse to the random port, until it needs new data (see figure 12.36).
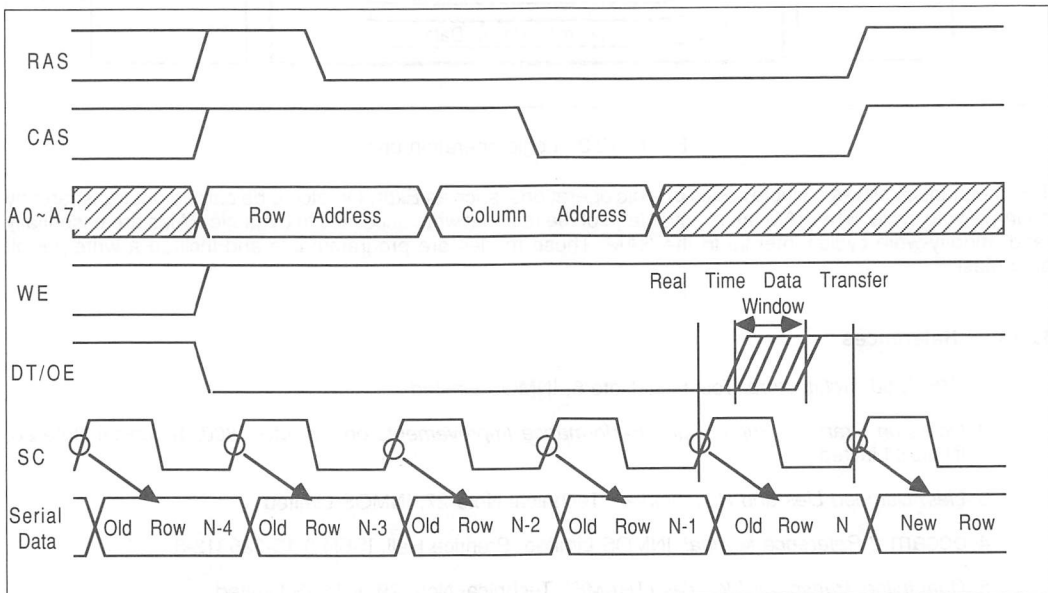
Figure 12.36 Video updating

The update cycle is the only time that the serial port and the random port interfere with each others operation, but because so much data is read into the internal register sets, this interference happens only occasionally,

ie. every 256 serial port access cycles. This means that a frame store directly mapped into a processors address map will use very little of the processors access to memory to refresh the display.

12.9.2   Video RAM logic operations

Some video RAMs have an internal logic operation unit (See figure 12.37). This unit can be set into particular modes by using a special CAS before RAS write cycle. The modes are selected by writing data to various locations using this special cycle. The data written is used as a write mask when writing subsequently to the RAM.
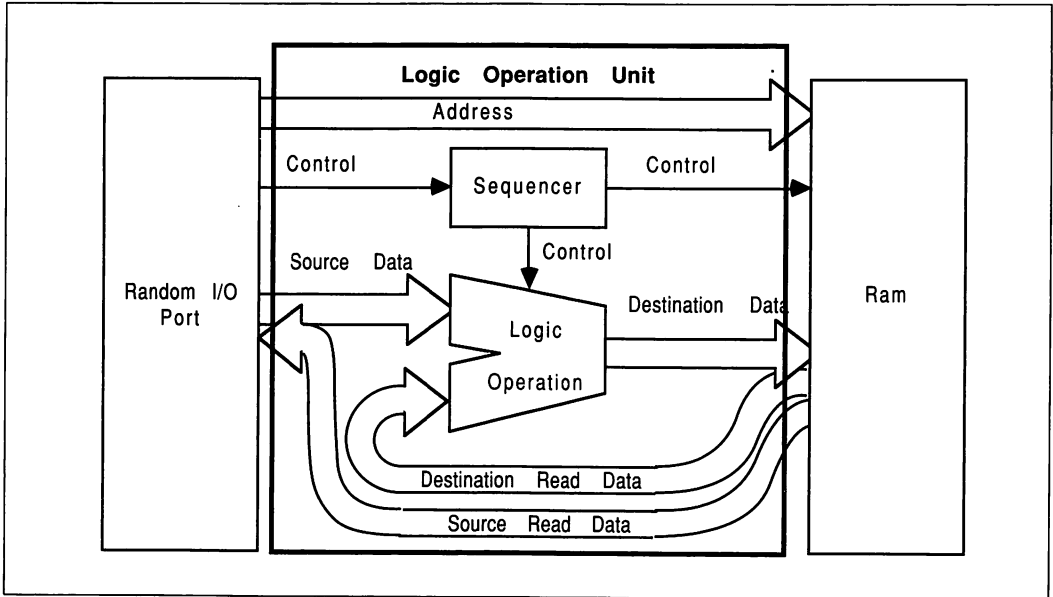
Figure 12.37 Logic operation unit

This mechanism allows a whole series of logic operations, such as Exor, Or, etc, to be carried out transparently during a write cycle. The RAM takes advantage of the fact that write accesses to dynamic RAMs are essentially read- modify-write cycles internal to the RAM. These modes are programmable and include a write-per-bit data mask.

12.10   References

1 *IMS T800 Architecture,* Technical Note 6, INMOS Limited

2 *Notes on Graphics Support and Performance Improvements on the IMS T800,* Technical Note 26, INMOS Limited

3 *Lies, Damned Lies and Benchmarks,* Technical Note 27, INMOS Limited

4 occam *2 Reference Manual,* INMOS Limited, Prentice Hall, ISBN 0-13-629312-3

5 *Dual Inline Transputer Modules (TRAMS),* Technical Note 29, INMOS Limited

6 *High Performance Graphics with the IMS T800],* Technical Note 37, INMOS Limited

7 *A Transputer Based Multi-User Flight Simulator,* Technical Note 36, INMOS Limited

Performance

13 Lies, damned lies and benchmarks

13.1 Introduction

A benchmark is supposed to be a standard measure of performance that enables one computer to be compared with another. However, a car is a simpler machine than a computer, and yet no-one expects all the relevant features of a car to be contained in a single number. Even in the specialised world of motor-racing, knowing the b.h.p. or the top speed is not enough to predict which car will be fastest round the track, and computing equivalents such as 'MIPS' or 'MFlops' are similarly misleading.

For any application it is performance *on that application* which counts, and benchmarks are relevant only so far as they resemble it. For example, some microprocessors can match the speed of super-minicomputers on non-numerical benchmarks, although their floating-point performance and input-output capability can be substantially inferior. Also, microprocessor architectures tend to give atypically high performance on small programs, by making good use of small register sets, caches, on-chip memory etc., and nearly all benchmark programs are very small in order to be easily disseminated.

Ideally, computers should be compared by running the intended application on each of them, but usually this is impractical, and benchmarks are often used instead. Some benchmarks have been carefully constructed and, in context, they can be a good guide to processor performance, provided their limitations are clearly understood. The Whetstone benchmark is one such, and is widely used as an indicator of performance on numerical tasks, although it omits some aspects of such applications, which we consider separately. The Savage benchmark tests only a narrow aspect of performance, but is often included in sets of benchmarks, so we consider it briefly. On the other hand, there are benchmarks which are badly constructed and cannot be related to any real application. An example is the Dhrystone benchmark, which, regrettably, is also widely used as a vague measure of processor power.

It is important to realise that all of these benchmarks are intended as tests for single-processor machines. None of them are particularly suited to parallelism; but then none of them are real application programs! Real programs are generally used to process data of some kind, and very often different parts of the data can be dealt with independently, allowing for large performance gains when several processors are used. Applications designed with parallelism in mind can often also be split into parts which can perform successive operations on the same flow of data in parallel, using a pipeline or other structure, allowing still more processors to be used effectively.

It is likely that the wide variety of possible architectures for parallel machines will render benchmarking impractical. Until that time we must live with benchmarks, so in this note we look at these three: the Whetstone, the Savage and the Dhrystone. We consider their merits and limitations, and provide performance figures and source listings.

13.2 The Whetstone benchmark

The Whetstone benchmark program [1] was constructed to compare processor power for scientific applications. Running the program is considered equivalent to executing (approximately) one million 'Whetstone' instructions. Performance, as measured by the benchmark, is quoted in 'Whetstones per second' and differs from any measure of pure floating-point performance given in 'flops'. In addition to floating-point operations, it includes integer arithmetic, array indexing, procedure calls, conditional jumps, and elementary function evaluations. These are mixed in proportions carefully chosen to simulate a 'typical' scientific application program of a decade ago.

13.2.1 Understanding the program

The virtue of the Whetstone benchmark is that it approaches real programs in complexity, whereas many other benchmarks only measure performance on simple loops. For example, a large part of the 'Linpack' benchmark effectively measures only the time to perform a loop of the form:

```
SEQ i = 0 FOR N
  a[i]:= b[i]+(t*c[i])
```

However, this complexity means that in order to relate the resulting performance figures to a real application, it is necessary to consider the precise composition of the benchmark. The occam source of the Whetstone is given in section 13.11. This is a straightforward translation of the ALGOL original, which consists of a series of modules designed to typify different aspects of a scientific computation. The core of each module is performed a certain number of times, determined by a 'best fit' to statistics of actual programs.

The time taken to execute a particular module may depend more on the speed of floating-point operations than on the specific task it represents. For example, module 2 is concerned with 'array accessing', but for each iteration of the loop there are 20 array accesses and 17 floating-point operations. On machines where the duration of a floating-point operation is much longer than the time taken to load or store a number, the floating-point operations will dominate the time to perform the module. This is also true of other modules. So the overall Whetstone performance will be largely determined by the floating-point speed of such machines. It will also depend on the speed of evaluation of elementary functions, because of the large number of such evaluations in modules 7 and 11. This is an area where applications vary widely, and the Whetstone represents an average which may be very different from any particular application.

13.2.2   The effect of optimisations

Since the benchmark is written in a high-level language (originally ALGOL; commonly FORTRAN; and in this case occam) it must be compiled before it can be executed. This makes the interpretation of the results more difficult since they depend not only on the hardware but also on the software which is used. As compilers become more sophisticated there is a danger that the original purpose of the benchmark will be lost in all the optimisations that can be done. The purpose of the benchmark is to cause the execution of (typically) one million 'Whetstone' instructions, which represent low-level operations of an abstract machine, and *not* to get through a particular FORTRAN program as fast as possible. Thus 'global' or source-level optimisations (either automatic or by hand) invalidate the benchmark since they miss out some of the 'Whetstone' instructions. Indeed, since no-one is interested in the results of the computations they could be optimised out altogether! By contrast the choice of high-level language to express the benchmark is relatively insignificant, provided its semantics are not too different from those of FORTRAN or ALGOL.

The occam compilers used to benchmark transputers aim to produce efficient code, but do not perform global or source-level optimisations. Consequently all the 'Whetstone' instructions implicit in the original program are performed.

13.2.3   Limitations of the Whetstone

It is important to realise that significant aspects of many contemporary scientific calculations are absent from the Whetstone, whilst others are over-emphasised:

  1 No consideration is given to the quality of floating-point calculations, and their speed is measured only indirectly.

  2 There are no multi-dimensional arrays, which are common in numerical programs, and the arrays which are present are very small.

  3 The number of elementary function evaluations is probably atypical of modern programs, and despite this heavy usage no account is taken of their accuracy.

We examine these points in the following sections.

Floating-point operations on the IMS T414 and IMS T800

Floating-point operations on the IMS T414

The floating-point operations provided for the IMS T414 are both fast and of high quality. Although the IMS T414 was designed to provide fast arithmetic operations on 32-bit integer values, it was appreciated that for many applications it would be necessary to perform floating-point arithmetic and so there are special instructions in the IMS T414 to support the implementation of floating-point operations in software.

The use of formal program proving methods has ensured that the quality of the software implementation is

very high [2]. The software packages correctly implement IEEE-standard floating-point arithmetic, *including the handling of denormalised numbers.*

Although implemented in software, floating-point operations on the IMS T414 are very fast, comparable with those performed by special floating-point co-processor chips. For example, the assignment in the occam fragment below:

```
REAL32 a, b, c :
SEQ
  . . .
  a := b * c
  . . .
```

will execute in about 11 μS, provided all the code and variables are in internal RAM. By comparison, the same assignment on an 8 Mhz Intel 80286/80287 combination would take about 31 μS (using the fastest possible memory). Even on 64-bit floating-point numbers, where it might be expected that software would lose out against hardware, the IMS T414 would take about 38 μS whilst the Intel combination would take about 44 μS.

Floating-point operations on the IMS T800

To achieve even higher performance than the IMS T414, the IMS T800 has a 64-bit floating-point unit on-chip. Its microcode was derived from the formally-proven occam implementation, so that the results of floating-point calculations by the two processors are identical (and correct) — only the speed differs. On an IMS T800 the assignment above would take only 29 cycles (1.45 μS for a 20MHz version, 0.97 μS for a 30MHz version), again assuming internal RAM is used.

The table below gives the typical and worst case operation times for floating point arithmetic on an IMS T414 (50 nS cycle time) and on an IMS T800 (50 nS and 33 nS cycle times). For the IMS T414 this assumes the code of the floating-point package is in the internal RAM.

Floating-point operation times

	IMS T414-20		IMS T800-20		IMS T800-30	
	Typical	Worst case	Typical	Worst case	Typical	Worst case
REAL32						
+, −	11.5 μS	15.0 μS	350 nS	450 nS	230 nS	300 nS
*	10.0 μS	12.0 μS	550 nS	900 nS	370 nS	600 nS
/	11.3 μS	14.0 μS	800 nS	1400 nS	530 nS	930 nS
REAL64						
+, −	28.2 μS	35.0 μS	350 nS	450 nS	230 nS	300 nS
*	38.0 μS	47.0 μS	1000 nS	1350 nS	670 nS	900 nS
/	55.8 μS	71.0 μS	1550 nS	2150 nS	1030 nS	1430 nS

Multi-dimensional arrays

Although not represented in the Whetstone benchmark, multi-dimensional arrays are common in many numerical applications. The IMS T414 and IMS T800 have a fast multiplication instruction ('product') which is used for the multiplication implicit in multi-dimensional array access. For example, in the following fragment of occam:

```
[20][20]REAL32 A :
SEQ
  . . .
  B := A[I][J]
  . . .
```

performing the assignment involves calculating the offset of element A[I][J] from the base of the array A.

The transputer compiler would generate the following code for this computation:

```
load local       I
load constant    20
product
load local       J
add
```

Since the **product** instruction executes in a time dependent on the highest bit set in its second operand, and the highest bit set in the constant **20** is bit 5, in this case the 'product' instruction will execute in only 8 cycles. In general, the multiplication in an address calculation is performed in a time approximately proportional to the logarithm of the array dimension. When combined with the concurrent operation of the CPU and FPU on the IMS T800 this enables address calculations to be entirely overlapped with floating-point calculations in most cases.

Elementary functions on the IMS T414 and IMS T800

The implementation of elementary functions involves a trade-off between speed, accuracy, and code-size. Whilst total accuracy is mathematically impossible, errors must be kept within reasonable bounds or else the functions are useless. The need to constrain code-size precludes the use of certain very fast algorithms which make use of very large look-up tables and linear interpolation.

The elementary function libraries used on the INMOS transputers are written in occam. They use rational approximations (quotients of polynomials), rather than table look-up or 'CORDIC' methods, as this gives the fastest execution whilst remaining accurate and code-compact. The single-length functions typically require a few hundred bytes of code (approximately 400 on the IMS T414 and 300 on the IMS T800), and have average errors of less than half a unit in the last bit. The functions handle all IEEE-standard values, including denormalised numbers, Not-a-Numbers, and Infinities. Further details are given in [3] and [4].

On the IMS T414 the rational approximations are computed using fixed-point arithmetic rather than floating-point. The IMS T414 has a 'fractional multiply' instruction which multiplies two 32-bit numbers together, treating each as a fraction between +1 and −1; the normal 'add' instruction will add such fractions. As a result of this the multiply and add, needed in each stage of a polynomial evaluation, will execute in under 3.5μS; if floating-point arithmetic were used these operations would take about seven times as long.

However the performance of the IMS T800 FPU is such that the multiply and add stage of a floating-point polynomial takes only 0.9 μS, so the library for this processor evaluates the rational approximations using floating-point arithmetic. Of course this library may be used on the IMS T414, producing identical results to those which would be obtained on an IMS T800, because of the equivalence of the floating-point software and hardware.

The importance of the speed of elementary function evaluation to the overall Whetstone performance figure is indicated by the proportion of time spent evaluating them, as indicated in the following table :

Percentage of total execution time

Processor :	IMS T414		IMS T800	
Floating-point format :	Single	Double	Single	Double
Trigonometric functions	26%	34%	23%	29%
Standard functions	13%	17%	21%	23%
Total	**39%**	**51%**	**44%**	**52%**

These percentages would probably be lower on a processor with special hardware for speeding up elementary function evaluation. Neither the IMS T414 nor the IMS T800 have any such special hardware, since including it would have compromised some other aspect of performance, so the speed and accuracy of elementary function evaluation is a good test of these processors. This is considered more fully in the next section, and timings for the individual functions are given in section 13.10.

13.3   The Savage Benchmark

13.3.1   Speed and accuracy of elementary functions

The Savage benchmark is a benchmark of elementary function evaluation only. It is actually named after its creator [5], although it is indeed quite a vicious test of an unsuspecting function library! It tests both speed and accuracy; in occam it is:

```
#USE dblmath
REAL64 a :
SEQ
   a := 1.0(REAL64)
   time ? start.time

   SEQ i = 0 FOR 2499
     a := DTAN(DATAN(DEXP(DALOG(DSQRT(a*a))))) + 1.0(REAL64)

   time ? finish.time
```

If the function subroutines were exact the final value of **a** would be 2500.0, so the difference from this figure is a measure of their accuracy. However it is important to note that the format (in this case IEEE double-precision) enforces a fundamental limitation no matter how carefully the functions are evaluated. The minimum error that can be achieved using double-precision floating-point is $1.177 * 10^{-9}$, and it can be seen from the table in section 13.11 that the occam function library produces a result which is very close to this figure. Some implementations give results more accurate than this, by using 'extended double precision' (80 bits) to evaluate the expression, only rounding to double-precision when the store into **a** is done.

Some results from this benchmark are given in section 13.8. It is certainly **not** typical of application programs, but it does give some indication of performance on elementary function evaluation only.

13.4   The Dhrystone benchmark

13.4.1   String manipulation performance

The Dhrystone [6] is a synthetic benchmark designed to test processor performance on 'systems programs'. In fact it has a number of flaws which seriously limit its usefulness as a guide to performance on 'typical' programs. Unfortunately its use has become widespread, with results published on the USENET, and manufacturers reporting their performance in terms of 'Dhrystones per second'. It was originally published in Ada, but the most widely used version is a translation into C, distributed over USENET.

As the construction of the Dhrystone is fully explained in the original publication, our discussion of the benchmark is limited to its drawbacks. The two principal flaws are the omission of any significant looping from the program and the inclusion of character string operations.

Whilst the Dhrystone's major advantage over many small benchmarks is that is does not consist of just a single loop, it suffers from the drawback that it does not do any significant amount of looping. This is unsound because most programs do contain loops and code executed within them will often account for most of the execution time. Also, when generating code for loops, a good compiler will seek to minimise the time to execute the loop repeatedly, possibly at the expense of more loop initialisation. Furthermore, research shows [7] that the code found within loops differs from code outside of loops; for example, most accesses to subscripted variables occur within loops.

The second major drawback of the Dhrystone that it uses strings, even though the only dynamic statistics in [6] show no use of strings (although the *static* statistics from the same source do show use of strings). In addition, the use of strings causes a large number of other problems with the benchmark. There are too many to consider in detail, so we will just look at the most significant.

The first problem comes from the method of construction of the benchmark, which was to ensure that the distribution of operators and operands matched that found in 'typical' programs. Unfortunately, the operators

and operands seem to have been treated independently, and as a result, the statement

> **if String_Par_In_1 > String_Par_In_2**

occurs in the Ada original. This may look inoffensive but when a translation into, for example, C occurs the result is

> **if (strcmp(StrParI1, StrParI2) > 0)**

which involves a very suspicious looking call to a library routine. As very little computation is performed in the benchmark this may be very significant. The amount of time taken to perform the comparison will, in fact, depend on the two strings being compared. In the Dhrystone the strings used are:

> **"DHRYSTONE PROGRAM, 2'ND STRING"**

and

> **"DHRYSTONE PROGRAM, 1'ST STRING"**

which match for the first 19 characters! The overall result of this is that, with a straightforward implementation of **strcmp** the only loop of any significance has been introduced by accident rather than by design.

The second problem is that the program contains a string assignment, which also becomes more blatant when the program is translated. In the Dhrystone as originally published, written in Ada, the strings in the program were declared to be 30 characters long. This means that a processor with the ability to copy data in blocks would be able to do the assignment very efficiently. When the translation to C takes place the translator has to make a choice; either the strings are converted into C strings, or they are changed into a structure. The former is more natural whilst the latter is more in keeping with the original program. The effect of this is, again, that a seemingly small part of the benchmark contributes significantly to the overall result.

One final point that should be noted is that the Dhrystone program, although intended to represent a typical 'system program', is actually extremely small, which again may make the results misleading.

The best known version of the Dhrystone benchmark is that in C, distributed on the USENET. It is a fair translation of the Ada except that it uses C-strings rather than fixed-sized byte arrays. The consequences of this alteration have already been discussed.

For some time an erroneous version of the Dhrystone was circulated on the USENET. When making comparisons of performance it is essential to check that the Dhrystone figure is for the correct version of the benchmark, known as version 1.1 by the USENET community. Figures for this erroneous version would be substantially higher than figures for the correct version. In particular the figures given in [8] are for the erroneous version.

The occam version attempts to be as close to the Ada as possible. There are some problems with this which were tackled as follows. The first difficulty is that the Ada Dhrystone uses structures, which occam does not support. The occam Dhrystone simulates structures using arrays, with the byte array (string) being 'punned' onto several words of the array. The second problem is that occam does not provide dynamic storage allocation which is used for allocating the structures. The occam Dhrystone uses an array of structures instead (this is of no significance to performance as the allocation of the structure is not timed as part of the benchmark). There are some other minor changes which have been necessitated such as re-ordering the declaration of procedures as in occam they must be declared before they are used.

The source of the occam version of the Dhrystone benchmark is given in section 13.12.

13.5   Conclusion

The Whetstone benchmark is one of the most respected and widely used measures of performance on 'scientific' applications, even though it does not address important aspects of such computations, and over-emphasises others. The IMS T414 and IMS T800 microprocessors are very well suited to such applications, and this is reflected in their Whetstone performance, shown in section 13.7.

The Savage benchmark only measures performance on elementary functions, but is quite widely used in the microcomputing world. Although Transputers have no special hardware for elementary functions, in order to maximise performance on more common operations [4], they perform extremely well, as can be seen from the results in section 13.8.

Thus the IMS T414 surpasses all other single-chip processors in performing numerical calculations with software, and outperforms many processor /co-processor combinations. The IMS T800 is the world's fastest microprocessor, superior even to multi-chip sets and bit-slice machines.

The Dhrystone is also widely used, even though it is essentially useless as an indicator of performance on real programs. The table in section 13.9 shows that Transputers give a high figure on this benchmark, but this is of relatively little significance. It is interesting to note that at least one recent 32-bit microprocessor has special hardware for processing strings; not surprisingly its projected Dhrystone figure is extremely high. However only programs that only process strings are likely to realise this promised performance. Transputers have not been optimised to 'pass' a particular benchmark; they are general-purpose processors delivering high performance on all applications.

13.6   References

1 *A Synthetic Benchmark*, Curnow H.J., and Wichmann B.A., Computer Journal 19 no. 1, February 1976.

2 *Formal Methods Applied to a Floating Point Number System*, Barrett G., Oxford University Computing Laboratory Technical Monograph PRG-58 1987.

3 *Transputer Development System Manual*, INMOS Limited, Prentice Hall 1988.

4 *Technical Note 6: IMS T800 Architecture*, INMOS Limited, Bristol, U.K. INMOS 1986.

5 *Dr. Dobb's Journal*, Savage B., September 1983, p120.

6 *Dhrystone: a synthetic systems programming benchmark*, Reinhold P. Weicker, Communications of the ACM, Vol. 27, Number 10, October 1984.

7 *An Empirical Analysis of FORTRAN programs*, Robinson and Torsun, Computer Journal 19 no. 1, February 1976.

8 *The 80386: A High Performance Workstation Microprocessor*, Intel Corporation, 1986, Order number: 231776-001.

13.7   Comparative Whetstone benchmark results

The following tables compare the performance figures of the transputers with other processors and processor /co-processor combinations for both the single and double precision Whetstone benchmarks. Some of the figures may have been superseded since these tables were compiled, but they are adequate for illustrative purposes.

System	Thousands of Single-precision Whetstones per Second
IMS T800-30 (projected)	6800
IMS T800-20	4548
WE 32200/32206-24	2800
INTEL 80386 + 80387	1860
VAX 11/780	1083
MVII	925
SUN-3	860
NS 32332/32081	728
IMS T414-20	704
NS 32032 and 32081	390
INTEL 286/287	300
IBM RT-PC + FPA	200
IMS T212-20	181
INTEL 8086 + 8087	178
MC 68000	13
IBM RT-PC	12

System	Thousands of Double-precision Whetstones per Second
IMS T800-30 (projected)	4400
IMS T800-20	2932
INTEL 80386 + 80387	1730
MVII	925
SUN-3	790
VAX 11/780	715
IMS T414-20	161
INTEL 8086 + 8087	152

Systems used for the benchmarks

IBM RT-PC	software only
IBM RT-PC + FPA	with NS32081 floating-point chip, in 'direct mode'
IMS T212-20	20 MHz internal clock rate, using product occam compiler
IMS T414-20	20 MHz internal clock rate, using product occam compiler
IMS T800-20	20 MHz internal clock rate, using product occam compiler
IMS T800-30	30 MHz internal clock rate, scaled from -20 result
INTEL 8086 + 8087	8 MHz
INTEL 286/287	10 MHz
INTEL 386/387	20 MHz
MC 68000	10 MHz, assembler coded software floating-point
MVII	MicroVAX II with FPA, running MicroVMS
NS 32032 and 32081	10 MHz
NS 32332 and 32081	15 MHz
SUN-3	MC 68020 (16 MHz) and MC 68881 (12.5 MHz)
WE 32200/32206-24	24 MHz
VAX 11/780	8MB memory, FPA, running under UNIX 4.3BSD

The figures for the IMS T414-20 were obtained by running the program on an IMS T414B-20 (50 nS cycle time), with 150 nS cycle time external memory. **Note that running the program on a slower system, such as are provided by INMOS for hosting the development system, will give a lower figure.** The figures for the IMS T800-20 were obtained by running the program on an IMS T800C-20 (50 nS cycle time). Figures for the faster version (30 MHz) were then obtained by straightforward scaling.

The figure for the IMS T212-20 was obtained by running the program on an IMS T212-20 (50 nS cycle time), with 100 nS cycle time external memory, using the technique of section 13.13.

Our sources for the other figures are as follows:

IBM RT-PC	IBM RT Personal Computer Technology, SA 23-1057, IBM 1986
INTEL 8086 + 8087	Sun-3 Benchmarks (Sun Microsystems, inc)
INTEL 286/287	Sun benchmark document
INTEL 386/387	Doug Rick, 80387 Marketing Manager
MC 68000	Published figure
MVII	Sun Benchmark document
NS 32032 and 32081	Ray Curry, National Semiconductor, via USENET
NS 32332 and 32081	Ray Curry, National Semiconductor, via USENET
SUN-3	Sun published data
WE 32200/32206-24	Electronics, December 18, 1986
VAX 11/780	John Mashey at MIPS Computer Systems, via USENET

13.8   Comparative Savage benchmark results

System	CPU	FPP	MHz	Language	Time (seconds)	Error (absolute)
IMS T800 (proj.)			30.0	occam	0.3	1.2E−9
IMS T800			20.0	occam	0.4	1.2E−9
Sun-3/160	68020	68881	16.67	Sun 3.0 FORTRAN 77	0.4	2.0E−12
HP 9000/320	68020	68881		Pascal	0.7	2.8E−7
VAX 8600				FORTRAN 77	0.9	1.8E−8
DMS	8086	8087		Turbo Pascal	3.8	1.1E−9
Zenith Z-248	80286	80287	8.0	FORTRAN 77	4.5	1.2E−9
IMS T414			20.0	occam	6.3	1.2E−9
IBM PC-AT	80286	80287	6.0	Turbo Pascal	7.4	1.2E−9
Sun-3/160	68020		16.67	Sun 3.0 FORTRAN 77	21.5	3.1E−7
IMS T212			20.0	occam	21.9	1.2E−9
Turbo-Amiga	68020		14.32	Absoft F77 V2.2B	21.9	1.8E−7

Information in this table (except for the Transputer figures) was supplied on USENET on 16th December 1986 by Al Alburto et al. The Transputer figures were obtained using the product occam compiler and libraries. The time for the IMS T800-30 was obtained by scaling the -20 result.

13.9   Comparative Dhrystone benchmark results

The following tables compare the performance of INMOS Transputers with other processors. The figure for the IMS T414 was obtained from an IMS B001 evaluation board, running an IMS T414B-20 with 3 cycle external memory. **Note that running the program on a slower system, such as are provided by INMOS for hosting the development system, will give a lower figure.** The other transputer figures were obtained by running the program on INMOS TRAMs.

System	Dhrystones per Second
IBM 3090/200	31250
IMS T800-30 (proj.)	13400
IMS T800-20	8956
IMS T212-20	8711
IMS T414-20	8193
VAX 8600	6423
Gould PN9080 Custom ECL	4992
Intel 386-16 (predicted)	4300
MC68020-17	3977
Intel 80286-9	1976
VAX 11/780	1650
MC68000-8	1136

It should be noted that Dhrystone figures, especially those quoted by manufacturers, are often invalid. Either they refer to the incorrect version 1.0 (and if no version is given, this is usually the case) or else they use optimising compilers, which are forbidden for this benchmark (frequently both). The figures above are believed

to be free of such contamination. It is regretted that no such figure is currently available for the 80386, and so an old predicted figure is given instead.

13.10   Elementary function performance

The table below gives the time taken to evaluate complete standard elementary functions on an IMS T800-20 and an IMS T414-20, each with 150 nS external RAM. Timings are given for both the case when the function code and the process workspace are in the on-chip RAM (for the IMS T800) and when the code is stored in the external RAM (both processors). The figures for each function were derived from measurements taken for 8000 arguments chosen at random from the interval [0.0, 10.0], except for arcsine and arccosine where the points were drawn from the interval [−1.0, 1.0], and the double-precision hyperbolic functions, for which the points were drawn from [0.0, 20.0].

Timings in microseconds

	IMS T800-20				IMS T414-20	
	ON-CHIP		OFF-CHIP		OFF-CHIP	
single-precision	mean	max	mean	max	mean	max
SQRT	5.9	6.4	6.0	6.5	26.0	27.6
ALOG	22.5	22.9	27.4	27.9	131.1	141.4
ALOG10	25.7	26.1	31.4	31.9	145.2	155.3
EXP	22.0	22.2	26.7	27.0	120.6	126.8
SIN	16.2	16.8	19.2	19.9	146.7	169.6
COS	18.9	19.3	22.2	22.6	178.1	186.8
TAN	18.4	19.2	22.3	23.2	142.7	164.4
ASIN	17.0	22.2	19.8	25.3	105.1	145.7
ACOS	16.7	21.3	19.8	24.8	101.5	132.6
ATAN	18.5	21.9	22.6	26.4	125.6	161.7
SINH	26.6	28.7	32.7	35.3	149.8	167.6
COSH	26.2	26.7	31.9	32.6	155.2	166.1
TANH	23.4	28.3	28.6	34.6	137.3	175.6
double-precision	mean	max	mean	max	mean	max
DSQRT	12.2	12.9	12.2	13.0	204.0	212.8
DALOG	34.5	38.5	45.0	46.0	607.9	636.1
DALOG10	42.5	43.1	49.9	50.9	658.7	687.4
DEXP	39.8	40.4	47.0	47.7	512.9	538.5
DSIN	33.1	34.2	38.1	39.2	590.0	655.2
DCOS	29.4	29.9	33.7	34.2	671.9	700.0
DTAN	35.8	37.3	42.0	43.7	632.4	712.2
DASIN	33.1	41.7	37.0	45.7	587.0	758.7
DACOS	32.9	40.6	36.8	44.9	574.3	714.2
DATAN	31.3	35.7	36.6	41.7	565.6	701.8
DSINH	45.9	47.8	54.2	56.5	609.5	649.0
DCOSH	44.8	45.4	53.3	54.1	618.4	648.5
DTANH	44.2	47.4	52.8	56.8	623.6	686.4

No figures are given for the IMS T212, but as a rough guide, consider single-precision functions to take between 5 and 7 times as long as for an IMS T414.

13.11   Source of the occam Whetstone program

This is the source of the occam version of the Whetstone benchmark. The output statments have been omitted, since they complicate the benchmarking process without affecting the results in any way. However the modules which are executed zero times have been included, since their omission would be a 'global optimisation' affecting the code-size. This is the single-precision version; the double-precision version is obtained by replacing all occurences of **REAL32** by **REAL64**, and all the library function calls by their double-precision versions.

```
PROC Whetstone (VAL [11]INT n, VAL INT iterations, INT time0, time1)

  #USE snglmath  -- this incorporates library code for the functions
  TIMER time :
  [4] REAL32 e1 :
  INT j, k, l :
  REAL32 t, t1, t2 :

  PROC p3 (VAL REAL32 xdash, ydash, REAL32 z)
    REAL32 x, y :
    SEQ
      x := t * (xdash + ydash)
      y := t * (x + ydash)
      z := (x + y) / t2
  :
  PROC p0 ()
    SEQ
      e1 [j] := e1 [k]
      e1 [k] := e1 [l]
      e1 [l] := e1 [j]
  :
  PROC pa ([4]REAL32 e)
    SEQ j = 0 FOR 6
      SEQ
        e[0] := (((e[0]    + e[1]) + e[2]) - e[3]) * t
        e[1] := (((e[0]    + e[1]) - e[2]) + e[3]) * t
        e[2] := (((e[0]    - e[1]) + e[2]) + e[3]) * t
        e[3] := ((((-e[0]) + e[1]) + e[2]) + e[3]) / t2
  :
  SEQ
    -- INITIALISE CONSTANTS
    t   := 0.499975(REAL32)
    t1  := 0.50025(REAL32)
    t2  := 2.0(REAL32)

    -- RECORD START TIME
    time ? time0

    -- MODULE 1 : SIMPLE IDENTIFIERS
    REAL32 x1, x2, x3, x4 :
    SEQ
      x1 :=  1.0(REAL32)
      x2 := -1.0(REAL32)
      x3 := -1.0(REAL32)
      x4 := -1.0(REAL32)
      SEQ i = 0 FOR n[0] * iterations
        SEQ
          x1 := (((  x1 + x2) + x3) - x4) * t
          x2 := (((  x1 + x2) - x3) + x4) * t
          x3 := (((  x1 - x2) + x3) + x4) * t
          x4 := ((((-x1) + x2) + x3) + x4) * t
```

```
-- MODULE 2 : ARRAY ELEMENTS
SEQ
  e1 [0] :=  1.0(REAL32)
  e1 [1] := -1.0(REAL32)
  e1 [2] := -1.0(REAL32)
  e1 [3] := -1.0(REAL32)
  SEQ i = 0 FOR n[1] * iterations
    SEQ
      e1[0] := (((e1[0]    + e1[1]) + e1[2]) - e1[3]) * t
      e1[1] := (((e1[0]    + e1[1]) - e1[2]) + e1[3]) * t
      e1[2] := (((e1[0]    - e1[1]) + e1[2]) + e1[3]) * t
      e1[3] := ((((-e1[0]) + e1[1]) + e1[2]) + e1[3]) * t

-- MODULE 3 : ARRAY AS PARAMETER
SEQ i = 0 FOR n[2] * iterations
  pa (e1)

-- MODULE 4 : CONDITIONAL JUMPS
SEQ
  j := 1
  SEQ i = 0 FOR n[3] * iterations
    SEQ
      IF
        j = 1
          j := 2
        TRUE
          j := 3
      IF
        j > 2
          j := 0
        TRUE
          j := 1
      IF
        j < 1
          j := 1
        TRUE
          j := 0

-- MODULE 5 : OMITTED IN ORIGINAL

-- MODULE 6 : INTEGER ARITHMETIC
SEQ
  j := 1
  k := 2
  l := 3
  SEQ i = 0 FOR n[5] * iterations
    SEQ
      j := (j * (k - j)) * (l - k)
      k := (l * k) - ((l - j) * k)
      l := (l - k) * (k + j)
      e1 [l - 2] := REAL32 ROUND ((j + k) + l)
      e1 [k - 2] := REAL32 ROUND ((j * k) * l)
```

```
-- MODULE 7 : TRIGONOMETRIC FUNCTIONS
REAL32 x, y :
SEQ
  x := 0.5(REAL32)
  y := 0.5(REAL32)
  SEQ i = 0 FOR n[6] * iterations
    SEQ
      x := t * ATAN ( (t2 * (SIN(x)*COS(x))) /
                      ((COS(x + y) + COS(x - y)) - 1.0(REAL32)) )
      y := t * ATAN ( (t2 * (SIN(y)*COS(y))) /
                      ((COS(x + y) + COS(x - y)) - 1.0(REAL32)) )

-- MODULE 8 : PROCEDURE CALLS
REAL32 x, y, z :
SEQ
  x := 1.0(REAL32)
  y := 1.0(REAL32)
  z := 1.0(REAL32)
  SEQ i = 0 FOR n[7] * iterations
    p3 (x, y, z)

-- MODULE 9 : ARRAY REFERENCES
SEQ
  j := 1
  k := 2
  l := 3
  e1 [0] := 1.0(REAL32)
  e1 [1] := 2.0(REAL32)
  e1 [2] := 3.0(REAL32)
  SEQ i = 0 FOR n[8] * iterations
    p0 ()

-- MODULE 10 : INTEGER ARITHMETIC
SEQ
  j := 2
  k := 3
  SEQ i = 0 FOR n[9] * iterations
    SEQ
      j := j + k
      k := j + k
      j := k - j
      k := (k - j) - j

-- MODULE 11 : STANDARD FUNCTIONS
REAL32 x :
SEQ
  x := 0.75(REAL32)
  SEQ i = 0 FOR n [10] * iterations
    REAL32 r2 :
    x := SQRT ( EXP (ALOG (x)/t1) )

-- RECORD FINISH TIME
time ? time1
:
```

Using the occam **Whetstone program**

The program given below will run the Whetstone benchmark twice; first to perform one million 'Whetstones', secondly to perform two million. The length of time taken to perform each run of the benchmark is sent on the channel **Out** to another process. This process should be running on another processor to avoid disturbing the Whetstone results.

The process connected to the other end of channel **Out** has to take the difference of the two times it is sent, and multiply the reciprocal by 10^{12} (because the time is for one million Whetstones, measured in micro-seconds). The result is then a measure of 'Whetstones per second', free from any bias introduced by irrelevant overheads.

```
PROC Benchmark (CHAN Out)

    ... SC Whetstone    -- the program in the previous section

    VAL [11]INT n IS [0, 12, 14, 345, 0, 210, 32, 899, 616, 0, 93]:
    -- n is the array of loop repetition counts
    INT time0, time1 :

    PRI PAR -- to get high-priority clock with 1us ticks
      SEQ
        Whetstone (n, 10, time0, time1)  -- one million whetstones
        Out ! time1 MINUS time0          -- output time difference
        Whetstone (n, 20, time0, time1)  -- two million whetstones
        Out ! time1 MINUS time0          -- output time difference
      SKIP -- null process to complete the PRI PAR construct
  :
```

The Whetstone benchmark is run at high priority to ensure that a 1 μS resolution timer is used.

The table **n** contains the number of iterations for each loop in the benchmark; these were calculated to make the benchmark equivalent to a 'typical' scientific application. This array of weights is an integral part of the benchmark, and if it is altered the results are **not** comparable with figures quoted in 'Whetstones'.

The actual number of iterations of each loop is the product of the table entry and the second parameter of the **Whetstone** procedure. If this is set to 10 then 1 million 'Whetstones' are performed.

13.12   Source of the occam **Dhrystone program**

This is the source of the program run on an IMS T414B-20, compiled with the product occam compiler.

```
PROC DHRYSTONE(CHAN OF INT32 In, Out)

  -- Define constants etc for the Struct equivalent
  VAL NULL IS 0 :
  VAL Ident1 IS 1 :
  VAL Ident2 IS 2 :
  VAL Ident3 IS 3 :
  VAL Ident4 IS 4 :
  VAL Ident5 IS 5 :

  VAL PtrComp    IS 0 :  -- 'pointer' to one of these records
  VAL Discr      IS 1 :
  VAL EnumComp   IS 2 :
  VAL IntComp    IS 3 :
  VAL StringComp IS 4 : -- StringComp is subsequent 30 bytes

  VAL StringSize IS 30 :
  VAL StringWords IS 8 : -- allocate 30/4 + 1 = 8 words on an IMS T414

  VAL StructSize IS StringWords + 4 :

  [3][StructSize]INT Records : -- all the records required

  -- Global variable declarations
  [51]INT     Array1 :
  [51][51]INT Array2 :
  INT         IntGlob :
  BOOL        BoolGlob :
  BYTE        Char1Glob, Char2Glob :
  INT         PtrGlb, PtrGlbNext :

  -- array placement
  PLACE Array1 AT (#800 / 4) : -- placement for an IMS T414 and IMS T800
  PLACE Array2 AT (#800 / 4) + 51 :
  Array2Glob IS Array2 :
  Array1Glob IS Array1 :

  INT FUNCTION Func1 (VAL BYTE CharPar1, CharPar2)
    INT Res :
    VALOF
      BYTE CharLoc1, CharLoc2 :
      SEQ
        CharLoc1 := CharPar1
        CharLoc2 := CharLoc1
        IF
          CharLoc2 <> CharPar2 -- true
            SEQ
              Res := Ident1
          TRUE
            Res := Ident2
      RESULT Res
  :
```

```
BOOL FUNCTION Func2 (VAL [StringSize]BYTE StrParI1, StrParI2)
  BOOL Res :
  VALOF
    INT FUNCTION strcmp (VAL [StringSize]BYTE S1, S2)
      INT order :
      VALOF
        IF
          IF i = 0 FOR StringSize
            S1[i] <> S2[i]
              IF
                (INT S1[i]) > (INT S2[i])
                  order := 1
                TRUE
                  order := -1
          TRUE
            order := 0
        RESULT order
    :
    -- StrParI1 = "DHRYSTONE, 1*'ST STRING"
    -- StrParI2 = "DHRYSTONE, 2*'ND STRING"
    INT IntLoc :
    BYTE CharLoc :
    SEQ
      IntLoc := 1
      WHILE IntLoc <= 1 -- executed once
        IF
          Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) = Ident1
            SEQ
              CharLoc := 'A'
              IntLoc := IntLoc + 1
          TRUE
            SKIP
      VAL CharLoc.int IS INT CharLoc : -- because no '>' for BYTEs
      IF
        (CharLoc.int >= (INT 'W')) AND (CharLoc.int <= (INT 'Z'))
          IntLoc := 7 -- not executed
        TRUE
          SKIP
      IF
        CharLoc = 'X'
          Res := TRUE -- not executed
        strcmp(StrParI1, StrParI2) > 0
          SEQ -- not executed
            IntLoc := IntLoc + 7
            Res := TRUE
        TRUE
          Res := FALSE
    RESULT Res
:
```

```
BOOL FUNCTION Func3(VAL INT EnumParIn)
  BOOL Res :
  VALOF
    INT EnumLoc :
    SEQ
      EnumLoc := EnumParIn
      IF
        EnumLoc = Ident3
          Res := TRUE
        TRUE
          Res := FALSE
    RESULT Res
:

PROC P8([51]INT Array1Par, [51][51]INT Array2Par,
                              VAL INT IntParI1, IntParI2)
  -- once; IntParI1 = 3, IntParI2 = 7
  INT IntLoc, IntIndex :
  SEQ
    IntLoc := IntParI1 + 5
    Array1Par[IntLoc]        := IntParI2
    Array1Par[IntLoc +  1] := Array1Par[IntLoc]
    Array1Par[IntLoc + 30] := IntLoc
    SEQ IntIndex = IntLoc FOR 2 -- twice
      Array2Par[IntLoc][IntIndex] := IntLoc
    Array2Par[IntLoc][IntLoc-1] := Array2Par[IntLoc][IntLoc-1] + 1
    Array2Par[IntLoc+20][IntLoc] := Array1Par[IntLoc]
    IntGlob := 5
:

PROC P7(VAL INT IntParI1, IntParI2, INT IntParOut) -- thrice
  -- 1) IntParI1 =  2, IntParI2 =  3, IntParOut :=  7
  -- 2) IntParI1 =  6, IntParI2 = 10, IntParOut := 18
  -- 3) IntParI1 = 10, IntParI2 =  5, IntParOut := 17
  INT IntLoc :
  SEQ
    IntLoc := IntParI1 + 2
    IntParOut := IntParI2 + IntLoc
:
PROC P5() -- once
  SEQ
    Char1Glob := 'A'
    BoolGlob := FALSE
:
PROC P4() -- once
  BOOL BoolLoc :
  SEQ
    BoolLoc := Char1Glob = 'A'
    BoolLoc := BoolLoc OR BoolGlob
    Char2Glob := 'B'
:
PROC P3(INT PtrParOut) -- executed once
  SEQ
    IF
      PtrGlb <> NULL -- true
        PtrParOut := Records[PtrGlb][PtrComp]
      TRUE
        IntGlob := 100
    P7(10, IntGlob, Records[PtrGlb][IntComp])
:
```

```
PROC P6(VAL INT EnumParIn, INT EnumParOut) -- once
  -- EnumParIn = Ident3, EnumParOut := Ident2
  SEQ
    EnumParOut := EnumParIn
    IF
      NOT Func3(EnumParIn) -- not taken
        EnumParOut := Ident4
      TRUE
        SKIP
    CASE EnumParIn
      Ident1
        EnumParOut := Ident1
      Ident2
        IF
          IntGlob > 100
            EnumParOut := Ident1
          TRUE
            EnumParOut := Ident4
      Ident3  -- this one chosen
        EnumParOut := Ident2
      Ident4
        SKIP
      Ident5
        EnumParOut := Ident3
:


PROC P2(INT IntParIO) -- executed once
  INT IntLoc, EnumLoc :
  BOOL Going :
  SEQ
    IntLoc := IntParIO + 10
    Going := TRUE
    WHILE Going -- executed once
      SEQ
        IF
          Char1Glob = 'A'
            SEQ
              IntLoc := IntLoc - 1
              IntParIO := IntLoc - IntGlob
              EnumLoc := Ident1
          TRUE
            SKIP
        Going := EnumLoc <> Ident1
:
```

```
PROC P1(VAL INT PtrParIn) -- executed once
  [StructSize] INT NextRecTemp :
  SEQ
    NextRecTemp := Records[PtrGlb]  -- must do this to avoid aliasing
    Records[PtrParIn][IntComp] := 5
    NextRecTemp[IntComp] := Records[PtrParIn][IntComp]
    NextRecTemp[PtrComp] := Records[PtrParIn][PtrComp]
    P3(NextRecTemp[PtrComp])
    -- NextRecTemp[PtrComp] = Records[PtrGlb][PtrComp] = PtrGlbNext
    IF
      NextRecTemp[Discr] = Ident1  -- it does
        INT IntCompTemp :
        SEQ
          NextRecTemp[IntComp] := 6
          P6(Records[PtrParIn][EnumComp], NextRecTemp[EnumComp])
          NextRecTemp[PtrComp] := Records[PtrGlb][PtrComp]
          IntCompTemp := NextRecTemp[IntComp]  -- to avoid aliasing
          P7(IntCompTemp, 10, NextRecTemp[IntComp])
      TRUE
        Records[PtrParIn] := NextRecTemp
    Records[Records[PtrParIn][PtrComp]] := NextRecTemp
:

PROC P0(INT32 out, VAL INT32 loops)
  TIMER TIME :
  [StringSize]BYTE String1Loc, String2Loc :
  INT IntLoc1, IntLoc2, IntLoc3 :
  BYTE CharLoc :
  INT EnumLoc :
  INT StartTime, EndTime, NullTime :
  VAL Loops IS 10 * (INT loops)
  SEQ
    -- initialisation
    -- initialise arrays to avoid overflow
    SEQ i = 0 FOR SIZE Array1Glob
      Array1Glob[i] := 0
    SEQ i = 0 FOR SIZE Array2Glob
      SEQ j = 0 FOR SIZE Array2Glob[0]
        Array2Glob[i][j] := 0
    PtrGlb := 1
    PtrGlbNext := 2
    -- initialise record 'pointed' to by PtrGlb
    Record IS Records[PtrGlb] :
    SEQ
      Record[PtrComp]  := PtrGlbNext
      Record[Discr]    := Ident1
      Record[EnumComp] := Ident3
      Record[IntComp]  := 40
      [4*StringWords]BYTE ByteBuff RETYPES
                     [Record FROM StringComp FOR StringWords] :
      [ByteBuff FROM 0 FOR StringSize]:=
                     "DHRYSTONE PROGRAM, SOME STRING"
    String1Loc := "DHRYSTONE PROGRAM, 1*'ST STRING"

    -- measure loop overhead
    TIME ? StartTime
    SEQ i = 0 FOR Loops
      SKIP
    TIME ? EndTime
    NullTime := EndTime MINUS StartTime
```

```
      TIME ? StartTime
      SEQ i = 0 FOR Loops
        SEQ
          P5()
          P4()
          -- Char1Glob = 'A', Char2Glob = 'B', BoolGlob = FALSE
          IntLoc1 := 2
          IntLoc2 := 3
          String2Loc := "DHRYSTONE PROGRAM, 2*'ND STRING"
          EnumLoc := Ident2
          BoolGlob := NOT Func2(String1Loc, String2Loc)
          -- BoolGlob = TRUE
          WHILE IntLoc1 < IntLoc2  -- body executed once only
            SEQ
              IntLoc3 := (5 * IntLoc1) - IntLoc2
              P7(IntLoc1, IntLoc2, IntLoc3)
              IntLoc1 := IntLoc1 + 1
          P8(Array1Glob, Array2Glob, IntLoc1, IntLoc3)
          -- IntGlob = 5
          P1(PtrGlb)
          SEQ CharIndex = INT 'A' FOR ((INT Char2Glob) - ((INT 'A')-1))
            -- twice
            IF
              EnumLoc = Func1(BYTE CharIndex, 'C')
                P6(Ident1, EnumLoc)
              TRUE
                SKIP
          -- EnumLoc = Ident1
          -- IntLoc1 = 3, IntLoc2 = 3, IntLoc3 = 7
          IntLoc3 := IntLoc2 * IntLoc1
          IntLoc2 := IntLoc3 / IntLoc1
          IntLoc2 := (7 * (IntLoc3 - IntLoc2)) - IntLoc1
          P2(IntLoc1)
      TIME ? EndTime

      out  := INT32 ((EndTime MINUS StartTime) - NullTime)
  :

  PRI PAR -- to get high priority timer
    INT32 count, result :
    SEQ
      In ? count
      P0(result, count)
      Out ! result
    SKIP
:
```

This program is intended to be run on a single processor, with channel **Out** mapped onto a hard link connected to another processor, running a process which outputs the number of loops to be performed (to improve the resolution of the timer) — typically 10000 — and then inputs the number of microseconds taken. A simple calculation turns this into a number of 'Dhrystones per second'.

13.13   Benchmarking the IMS T212

It should be noted that obtaining benchmark figures for the IMS T212 is slightly more involved than for either the IMS T414 or the IMS T800. This is because the built-in timer has only 16 bits on this processor, as opposed to 32 on the other two processors, so consequently the clock 'wraps round' very much faster. In

fact it does so faster than a benchmark program can be run, and so the run-time of the program cannot be obtained simply by reading the clock at the beginning and end of the run, as shown in the preceeding listings.

The solution to this problem is to use another processor to perform the timing. Instead of reading the timer the program on the IMS T212 sends a message to another processor (an IMS T414 or an IMS T800) which responds by reading its own timer. The quoted benchmark results for the IMS T212 were obtained in this way.

14 Performance maximisation

14.1 Introduction

The INMOS transputer family [1] is a family of microcomputers with high-performance processor, memory and communication links on a single chip, figure 14.1. The links are used to connect transputers together, and very large concurrent systems can be built from collections of transputers communicating via their links.
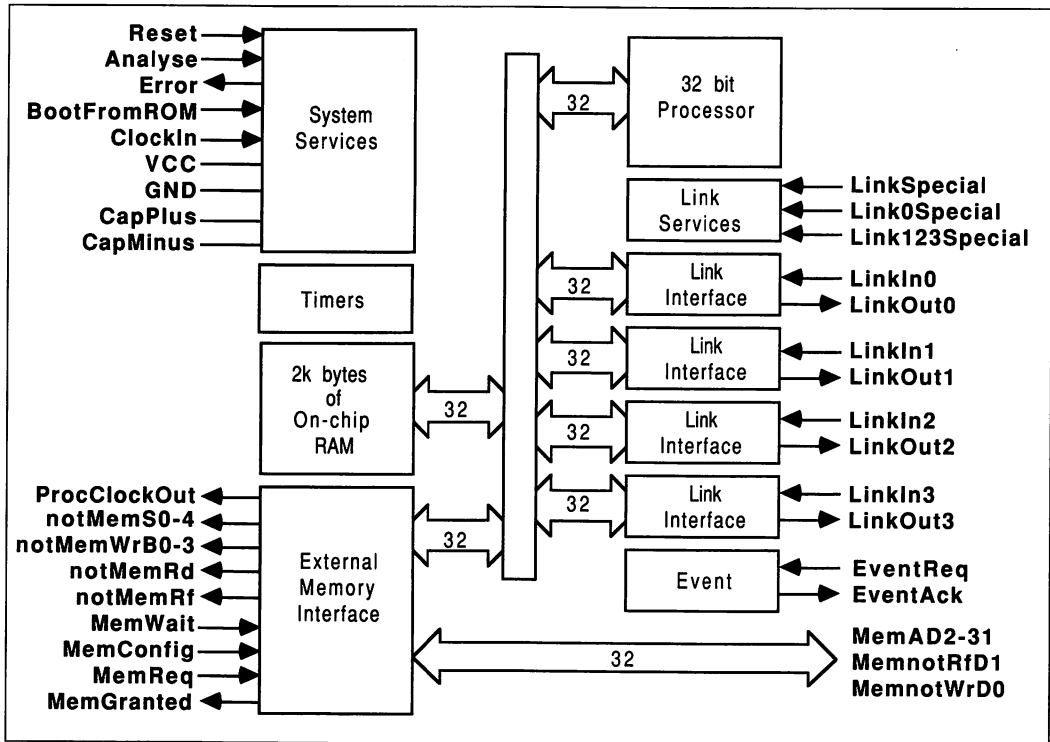

Figure 14.1 Transputer architecture

The occam programming language [2] was developed by INMOS to address the task of programming extremely concurrent systems. This document will illustrate how best to arrange occam programs in order to maximise the performance of transputer systems, with particular reference to the author's ray-tracing program [3].

All these performance enhancement techniques have been implemented in the ray tracer, and their use will be illustrated by fragments from this program.

Several topics will be discussed, falling into two main categories — maximising the performance of an individual transputer, and maximising the performance of arrays of transputers.

Note that all occam examples conform to the product release of the Transputer Development System.

14.2 Maximising performance of a single transputer

The following sections describe how to maximise the performance of a single transputer. However, all these performance maximisation techniques are highly relevant to maximising the performance of each processor in a multiple transputer system.

14.2.1   Making use of on-chip memory

To achieve maximum performance from a transputer it is important that good use is made of on-chip memory. On the IMS B004-4 Transputer Evaluation Board for example [4], the internal memory cycles in 66ns, whereas the off-chip memory cycles in 330ns. This factor of five degradation in memory speed can be reflected in program performance if heavily accessed locations are in off-chip memory.

On-chip memory is better used for scalar values and pointers rather than code and arrays. The IMS T414 fetches instructions in 32-bit words, so every code fetch cycle will pull in 4 instructions. Hence code accesses generally occur less frequently than data accesses. Also, every access to a data structure requires two or more scalar values and pointers to be accessed to determine the address of a component of the array.

Memory layout

The occam compiler and transputer loader software try to place scalar values and pointers on-chip. Three areas of store are allocated starting from the lowest free location in on-chip memory.

The first area holds the process workspaces; this is normally placed in on-chip memory. The second holds the program code; this is placed above the workspaces and most of it will be in off-chip memory. The third area holds the arrays; this is nearly always in off-chip memory (figure 14.2).
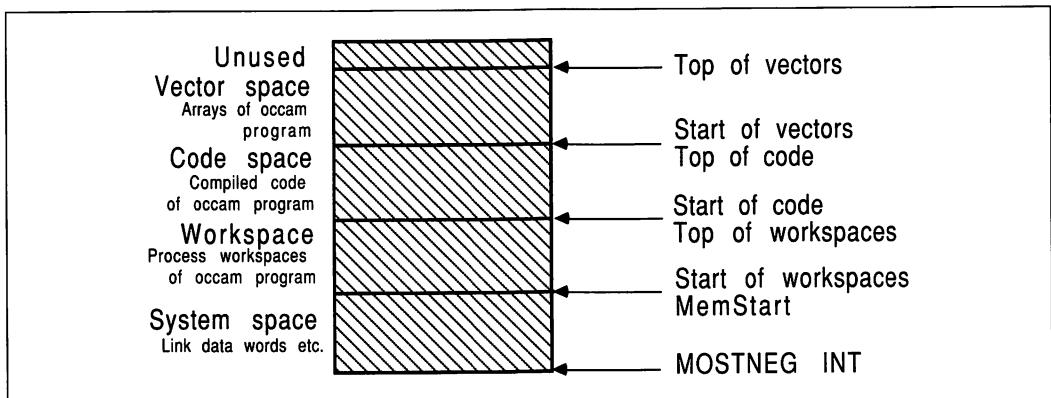

Figure 14.2 Memory layout of occam program

This is made possible because all data allocation in occam is static, and after compilation the loader knows exactly the data space requirement of the program. (Static allocation has one major drawback — recursion is not allowed in occam. Handling recursive algorithms in occam is described in section 14.7.)

If a program has a data space requirement of more than 4K bytes (the on-chip memory space of the IMS T800), then some data will be placed in off-chip memory. It is then up to the programmer to arrange his occam program such that the most frequently used variables are placed on-chip. The following sections will describe how to write occam programs which optimise use of on-chip memory.

Workspace layout

On the transputer, variables are accessed relative to a workspace pointer register, w [1]. Each occam process has its own workspace — a procedure call will generate a new workspace for the called procedure, and forking a set of parallel processes will generate a new workspace for each new process.

To maximise performance it is important that variables within the most frequently active workspace areas be in on-chip memory.

Workspace layout of called procedures

In occam, workspace for called procedures is allocated as a falling stack. Called procedures have their workspace placed at lower addresses than the caller. Scalar variables and pointers are located within the workspace. Arrays are normally located in the seperate off-chip storage area, but can be placed within the workspace if it is important that they are accessed rapidly.

The occam compiler places the most recently declared variables in the lowest workspace slots. For example, the following piece of code:

```
INT32 a, b, c :
[200] INT32 Vector :
PLACE Vector IN WORKSPACE:
SEQ
   a := 42
   b := #DEFACED
   c := #DEAF
   SEQ i = 0 FOR 200
      Vector [i] := 0
```

would result in the following workspace layout:

Variable	Workspace Location
a	205
b	204
c	203
Vector	2 .. 202
i	0 .. 1 (replicators consume 2 workspace slots)

Note that the replicator variable is implicitly declared last, and therefore takes up the two lowest workspace slots. However, **a, b** and **c** have ended up above the array **Vector**, and prefixing instructions are required to access them. If **a b** or **c** are going to be accessed frequently, it is better to declare them **after Vector**.

A procedure may access global variables and arrays; these will have been declared in an enclosing procedure. Global variables are accessed using a pointer in the procedure workspace. This pointer is the head of a list known as the *static chain* through which the procedure can access variables from the workspace of any enclosing procedure. To avoid lengthy access times and bulky code due to static chaining, frequently accessed global variables and vectors should be brought into local scope, either by passing them as parameters, or abbreviating them locally.

Further use of abbreviations to improve performance is discussed in 14.2.2.

Workspace layout of parallel processes

Workspace for parallel processes is allocated below the workspace of the parent. The first member of the **PAR** list is allocated workspace immediately below the parent, the second immediately below that, etc.
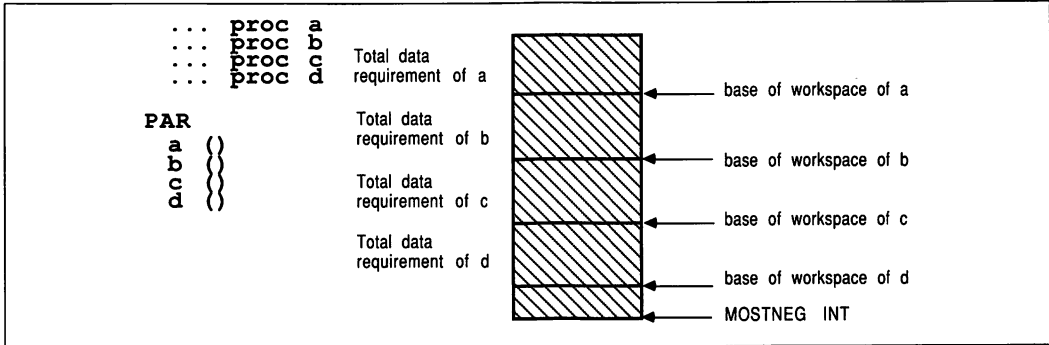

Figure 14.3 Workspace layout of parallel processes

If, in the example above any of the processes **a b c** or **d** were consuming large amounts of workspace, then the workspace of the others could be resident off-chip.

14.2.2   Abbreviations

Abbreviations are a powerful feature of the occam language. They can be used to bring non-local variables down into local scope, thus removing static chaining and speeding up access. They can also speed up execution by removing range check instructions. Where appropriate, **VAL** abbreviations should be used; for scalar values this creates a local copy of a varible rather than a pointer to it.

Abbreviations – removing range-checking code

By abbreviating sub-vectors of larger vectors and using constants to index into the sub-vector, the compiler will generate range-checking code for the abbreviation, but will not need to generate range-checking code for accesses to the sub-vector.

As an example of abbreviations removing range check instructions, here are two versions of the same procedure. Part of the ray-tracer, this procedure is initialising fields in a new node to be added into a tree. The identifier **nodePtr** points to the start of the node. The second version uses abbreviations, generates no range checking code (apart from initial generation of the abbreviation) generates shorter code sequences for each assignment, and executes more quickly.

```
PROC initNode ( VAL INT nodePtr )
  SEQ
    tree [ nodePtr + n.reflect] := nil
    tree [ nodePtr + n.refract] := nil
    tree [ nodePtr +    n.next] := nil
    tree [ nodePtr +  n.object] := nil
:
PROC initNode ( VAL INT nodePtr )
  node IS [ tree FROM nodePtr FOR nodeSize ] :
  SEQ
    node [ n.reflect] := nil
    node [ n.refract] := nil
    node [    n.next] := nil
    node [  n.object] := nil
:
```

Even if range-checking were switched off, the second version will execute more quickly. Without range check instructions, the statement   `tree [ nodePtr + n.refract] := nil`   will generate the following transputer instructions:

```
ldc     nil          -- get data to save
ldl     nodePtr      -- get pointer to base of node
ldl     static       -- get static chain
ldnlp   tree         -- generate pointer to tree ( in outer scope)
wsub                 -- generate pointer to tree [ nodeptr]
stnl    n.refract    -- and store to tree [ nodePtr + n.refract]
```

whereas the second version   `node [ n.refract] := nil`   will generate the following, appreciably shorter and faster fragment of code:

```
ldc     nil          -- get data to save
ldl     node         -- load abbreviation
stnl    n.refract    -- and store
```

Of course there is an initial overhead to generate the abbreviation, but this is rapidly swamped by the subsequent savings.

Abbreviations – opening out loops

Using abbreviations to open out loops can speed up execution considerably. Take the following piece of occam, a simple vector addition:

```
SEQ i = 0 FOR 20000
  a[i] := b[i] + c[i]
```

The transputer loops in about a microsecond, but adds in about 50 nanoseconds. Therefore to increase performance we must increase the number of adds per loop:

```
VAL bigLoops IS 2000 >> 4 :                    -- 2000 / 16
VAL leftOver IS 2000 - (bigLoops TIMES 16) :
SEQ
  SEQ i = 0 FOR bigLoops
    VAL base IS i TIMES 16 :
    aSlice IS [ a FROM base FOR 16 ] :
    bSlice IS [ b FROM base FOR 16 ] :
    cSlice IS [ c FROM base FOR 16 ] :
    SEQ
      aSlice  [0]  := bSlice [0]  + cSlice [0]
      aSlice  [1]  := bSlice [1]  + cSlice [1]
      aSlice  [2]  := bSlice [2]  + cSlice [2]
              ............
      aSlice [14]  := bSlice[14]  + cSlice[14]
      aSlice [15]  := bSlice[15]  + cSlice[15]
  SEQ i = 2000 - leftOver FOR leftOver
    a[i] := b[i] + c[i]
```

Obviously, loops can be opened out in any language, on any processor, and performance will tend be improved at the expense of increased code size. However, opening loops out in slices of 16 has a knock-on effect on the transputer, as optimal code **with no prefix instructions** is generated for each addition statement. Compare the code generated for the two statements:

```
a[i] := b[i] + c[i]

ldl   i
ldl   b
wsub
ldnl  0
ldl   i
ldl   c
wsub
ldnl  0
add
ldl   a
ldl   i
wsub
stnl  0

aSlice[15] := bSlice[15] + cSlice[15]

ldl   bSlice
ldnl  15
ldl   cSlice
ldnl  15
add
ldl   aSlice
stnl  15
```

The second piece of code is just over half the size of the first and the number of loop end (`lend`) instructions executed is reduced by a factor of 16.

14.2.3   Placing critical vectors on-chip

As mentioned above, in is sometimes important to place arrays in on-chip memory. For example, the following piece of code clears the screen of the IMS B007 graphics board [5]:

```
PROC clearScreen ( VAL BYTE pattern )
  -- the screen is declared as
  -- [2][512][512] BYTE screenRAM :
  [256][1024] BYTE screen RETYPES screenRAM [ currentScreen] :
  [1024] BYTE fastVec :   -- this is in on-chip memory
  PLACE fastVec IN WORKSPACE:
  SEQ
    initBYTEvec ( fastVec, pattern, 1024 ) -- fast byte initialiser
    SEQ y = 0 FOR 256
      screen [y] := fastVec
  :
```

This process fires off 256 block move instructions, each of 1024 bytes. Since the block move is reading from on-chip memory and writing to off-chip memory it will proceed more quickly than:

```
PROC clearScreen ( VAL BYTE pattern )
  [512*512] BYTE screen RETYPES screenRAM [ currentScreen] :
  initBYTEvec ( screen, pattern, 512*512 ) -- fast byte initialiser
  :
```

where all data accesses are to off-chip memory. The time saved during the block moves outweighs the cost

of setting up the parameters to the block moves, and of the initial `initBYTEvec`. See section 14.2.4 for more about block moves, and the source of `initBYTEvec`.

Beware the PLACE statement

A common mistake in trying to make occam go faster is to physically place data on-chip, using a **PLACE** statement. This does the right thing — the compiler will physically place the variable on-chip, but the variable will be outside local workspace.

Therefore to access the variable, its physical address must be generated, and an indirection performed to load the contents of the address.

For example, declaring a variable at word address 30 above MOSTNEG INT, and setting its value to 3:

```
INT a :
PLACE a AT 30 :   -- 30th word address above mint
a := 3

ldc   3
mint
stnl  30
```

This code sequence takes 6 cycles (300 ns on an IMS T414-20). Were **a** a local variable, the code sequence would be:

```
ldc   3
stl   a
```

and would take only 2 cycles (100 ns) if the workspace were on-chip.

Placing variables in on-chip memory can also be extremely dangerous; if the **PLACE**d variable accidentally overlays a workspace location the results will be unpredictable and could be disastrous.

The key to making variable accesses go faster is to **keep the workspace on-chip**. Then if it is necessary for a vector to be on-chip, it can be declared in local scope and placed in the workspace.

14.2.4   Block move

The IMS T414 vector assignment instruction **move** [1] is directly supported by the occam language. The vector assignment statement:

```
[65536] BYTE bigVec, otherVec :
[ bigVec FROM 0 FOR 65536] := [ otherVec FROM 0 FOR 65536]
```

compiles down to only 4 instructions:

```
ldl bigVec     -- assuming the vectors are abbreviated
ldl otherVec   -- locally
ldc 65536      -- this will be prefixed of course
move
```

A very fast vector initialiser can be written using block moves.

```
PROC initBYTEvec ( [] BYTE vec, VAL BYTE pattern, VAL INT bytes )
  INT dest, transfer :
  SEQ
    transfer := 1
    dest      := transfer
    vec [0]   := pattern
    WHILE dest < bytes
      SEQ
        [vec FROM dest FOR transfer] := [vec FROM 0 FOR transfer]
        dest      := dest + transfer
        transfer := transfer + transfer
  :
```

This performs a series of assignments of increasing length, initialising the first byte of the vector, then the next 2, then the next 4, 8, 16 etc. As printed above it will only initialise vectors which are an exact power of two in size, but very slight modifications make it completely general.

14.2.5   Retyping – accelerating byte manipulation

Under certain circumstances retyping can be used to speed up byte manipulation. If it is necessary to frequently extract byte fields from a word, then accessing retyping the word to a byte array is faster than shifting and masking. For example:

```
INT word :
[4] BYTE bWord RETYPES word :
SEQ
    ...  use bWord[0], bWord[1], bWord[2], bWord[3]
```

To access bits 16..23 in `word`, simply reference `bWord[2]`, which will generate:

```
ldc    2
ldlp   bWord   -- load base of bWord
bsub           -- select byte 2
lb             -- and load it
```

To perform byte operations on large arrays it is worthwhile moving portions of the array to a local (on-chip) array; this is because a block move transfers words and is therefore much faster than accessing individual bytes from an off-chip array. For example:

```
[1024] INT vector :
[] BYTE bytevector RETYPES vector :

[16] BYTE local :
PLACE local IN WORKSPACE :
INT base :
SEQ
  base := 0
  SEQ i = 0 FOR 64
    SEQ
      local := [bytevector FROM base FOR 16]
      base := base + 16
      SEQ i = 0 FOR 16
        SEQ
          ...  use local[i] to access each byte
```

14.2.6   Use TIMES

The IMS T414 transputer has a fast (but unchecked) multiply instruction, which is accessed with the occam operator TIMES. An integer multiply on the IMS T414-20 takes over a microsecond — using TIMES this will take as many processor cycles as there are significant bits in the right-hand operand, plus 2 cycles overhead. Therefore,

```
     a * 4
```

still takes over a microsecond, whereas

```
     a TIMES 4
```

takes only 6 cycles (300 ns). Therefore, when multiplying integers by small constants, use TIMES. Note that the IMS T800 Floating Point Transputer has a modified version of TIMES which optimally multiplies small negative integers.

14.3   Maximising multiprocessor performance

The following sections will describe how to obtain more performance from an array of transputers. However, only very general guidelines can be offered. Maximising multiprocessor performance is still an area of active research, and any solution will tend to be specific to the problem at hand.

14.3.1   Maximising link performance

The transputer links are autonomous DMA engines, capable of transferring data bidirectionally at up to 20 Mbits/sec. They are capable of these data rates without seriously degrading the performance of the processor. To achieve maximum link throughput from a multi transputer system the links and the processor should all be kept as busy as possible.

Decoupling communication and computation

To avoid the links waiting on the processor or the processor waiting on the links, **link communication should be decoupled from computation**.

For example, the following program is part of a pipeline, inputting data, applying a transformation to each data item, then outputting the transformed data:

```
     PROC transform ( CHAN in, out )
       [dataSize] INT data :
       WHILE TRUE
         SEQ
           in  ? data
           applyTransform ( data )
           out ! data
     :
```

If the channels in and out are transputer links, then the performance of the pipeline will be degraded. The SEQ contruct is forcing the transputer to perform only one action at a time; it is either inputting, computing or outputting; it could be doing all three at once. Embedding the transformer between a pair of buffers will improve performance considerably:

```
     PAR
       buffer ( in, a )
       transform ( a, b )
       buffer ( b, out )
```

The buffers are decoupling devices, allowing the processor to perform computation on one set of data, whilst concurrently inputting a new set, and outputting the previous set.

In this example the buffer processes will simply input data then output it. There is a transfer of data here which can be avoided, as all the data can be passed by reference:

```
[dataSize] INT a, b, c :
...   proc input
...   proc transform
...   proc output
SEQ
  input ( a)                    -- start-up sequence .. pull in data
  PAR
    input ( b)                  -- now transform that data
    transform ( a)              -- and pull in more ...
  WHILE TRUE
    SEQ                         -- and from here on
      PAR                       -- the buffers pass round-robin
        input      ( c)         -- between the inputter, transformer
        transform ( b)          -- and outputter
        output     ( a)
      PAR
        input      ( a)
        transform ( c)
        output     ( b)
      PAR
        input      ( b)
        transform ( a)
        output     ( c)
```

Instead of input and output operations transferring data between the processes, the processes transfer themselves between the data, each process cycling between the vectors a,b and c as the **PAR** statements close down and restart.

This is a special case, a data flow architecture where all communication and processing is synchronous — there is a lock-step **in, transform, out** sequence which allows this sequential overlay of computing and communication. This is not the case in many programs, where buffer processes are required.

Some applications are sufficiently concurrent that implicit buffering is taking place in processes which communicate directly with links. This is the case with the ray-tracer. The ray-tracer has extensive data routing processes, and the insertion of additional buffering processes unexpectedly reduced the performance (albeit by much less than one per cent). However these buffer processes have been shown to be important, as subtle deadlocks can occur if the buffers are removed.

Prioritisation

Correct use of prioritisation is important for most distributed programs communicating via links. If a message is transmitted to a transputer and requires throughrouting, it is essential that the transputer input the message then output it with minimum delay — another transputer somewhere in the system could be held up, waiting for the message. In such cases it is important to **run the processes which use the links at high-priority**. There will tend to be more than one process talking to links, at most eight, and the **PRI PAR** statement allows only one process at each priority level. It is necessary to gather together all the link communication processes, unify them into a process with a **PAR** statement, and run this process at high-priority.

The program from above now becomes:

```
[dataSize] INT a, b, c :
... proc input
... proc transform
... proc output
SEQ
  input ( a)              -- start-up sequence .. pull in data
  PRI PAR
    input      ( b)       -- now transform that data (HI-PRI)
    transform ( a)        -- and pull in more ...
  WHILE TRUE
    SEQ                   -- and from here on
      PRI PAR             -- the buffers pass round-robin
        PAR
          input    ( c) -- between the inputter, transformer
          output   ( a)
        transform ( b) -- and outputter
      PRI PAR
        PAR
          input    ( a)
          output   ( b)
        transform ( c)
      PRI PAR
        PAR
          input    ( b)
          output   ( c)
        transform ( a)
```

As an example, this is the outermost level of the `calculate` process in the ray tracer. Note the use of prioritisation, and global vectors. Everything is prioritised **except** the process performing the computation — a scheme which at first sight appears to be counter intuitive, but is of fundamental importance in a parallel system. Accidental or misguided prioritisation of computing processes will lead to disastrous performance degradation.

```
PROC calculate ( CHAN fromPrev, toNext, fromNext, toPrev,
                 VAL BOOL propogate )
  ... proc render
  ... proc routeWork
  ... proc mixPixels
  CHAN toLocal, fromLocal, requestWork :

  -- run all through routers at hi-PRI, and do
  -- all the floating point maths at lo-PRI

  [256] INT buffA, buffB :
  [(treeSize + worldModelSize) + gridSize] REAL32 heap :
  WHILE TRUE
    PRI PAR
      PAR
        routeWork  ( buffA, fromPrev, toNext, toPrev, local,
                        requestWork, propogate )
        mixPixels  ( buffB, fromLocal, fromNext, toPrev, buffers )
      render ( heap, toLocal, fromLocal, requestWork )
  :
```

14.3.2   Large link transfers

Setting up a transfer down a link takes about about a microsecond (20 processor cycles), but once that transfer is started it will proceed autonomously from the processor, consuming typically 4 processor cycles every 4 microseconds (one memory read or write cycle per 32-bit word). **Keep messages as long as possible.** For example:

```
[300] INT data :
SEQ
   out ! some.data; 300; [ data FROM 0 FOR 300]
```

is far better than

```
[300] INT data :
SEQ
   out ! some.data; 300
   SEQ i = 0 FOR 300
      out ! data [i]
```

However, long link transfers increase latency when data must be throughrouted. Some optimal message length will give the best compromise between overhead on setting up transfers, and overhead on throughrouting. A detailed discussion can be found in [6].

14.4   Dynamic load balancing and processor farms

Processor farms [7] are a general way of distributing problems which can be decomposed into smaller independent sub-problems. If implemented carefully, processor farms can give linear performance in multi transputer systems — that is ten processors will perform 10 times as well as one processor. Processor farms come into their own when solving problems where the amount of computation required for any given sub-problem is not constant.

For example, in the ray tracer one pixel may only require one traced ray to determine its colour, but other pixels may require over a hundred.

Rather than give each processor say one tenth of the screen (assuming ten processors in the array) , the screen is split into much smaller areas — in this case 8x8 pixels, giving a total of 4096 work packets for a 512x512 pixel screen. These are handed out piecewise to the farm. Each processor in the farm computes the colours of the pixels for that small area, and passes the pixels back, the pixel packet being an implicit request for another area of screen to be rendered.

Since work is only given to the farm on demand, load is balanced dynamically, with the whole system keeping itself as busy as possible. Buffer processes overlay data transfer with communication, reducing the communication overhead to zero, and the end-case latency of a processors farm implemented this way is far lower than in a statically load-balanced system.

Here is a diagram of the ray tracer.

The key to the processor farm is a valve process, allowing work packets into the farm only when there is an
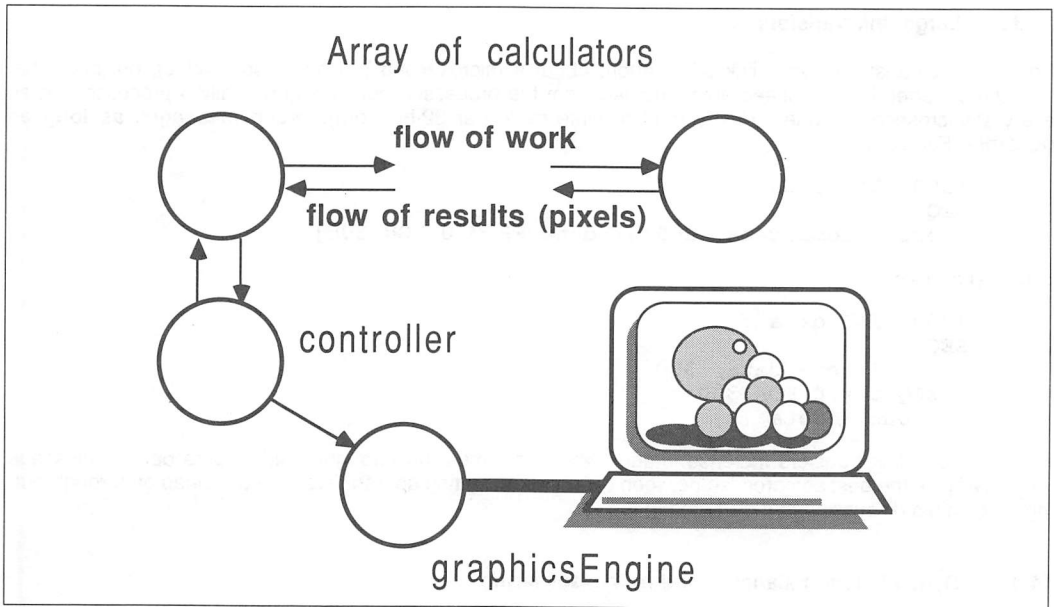
Figure 14.4 Structure of ray tracing program

idle processor. The structure of this valve is:

```
PAR
  -- pump work unconditionally
  SEQ i = 0 FOR workPackets
    inject ! packet
  -- regulate flow of work into farm
  SEQ
    idle := processors
    WHILE running
      PRI ALT
        fromFarm ? results
          idle := idle + 1
        (idle > 0) & inject ? packet
          SEQ
            tofarm ! packet
            idle := idle - 1
```

where the crucial statement is the guarded **ALT**,

```
        (idle > 0) & inject ? packet
```

only allowing work to pass from the pumper into the farm when there is an idle processor. The **ALT** is prioritised to accept results — this is explained in section 14.5.3.

The processor farm technique has been used to implement a very fast Mandelbrot Set generator [7, 8] and a step-coverage simulator for VLSI circuits [9]. A large forecasting/statistical modelling package is in the process of being implemented as a processor farm. In all cases fully implemented, linearity of performance to number of processors has been high, from 80–99.5%. That is, ten processors perform between 8 and 9.95 times as well as one processor.

14.5   A worked example : the INMOS ray tracer

Ray tracing [10] is a computer graphics technique capable of generating extremely realistic images. It handles inter-object reflections, refraction and shadowing effects in a simple and elegant algorithm. However, ray tracing has one major drawback — it devours computing resource. In [10] very simple scenes were rendered on a powerful minicomputer, taking from 45 to 180 minutes per image.

The structure of the INMOS ray tracer was described in [3] and [7] — in this section the performance enhancement techniques described above will be illustrated with reference to the ray tracer.

Finally, results will be presented comparing the optimised implementation of the ray tracer with deliberately de-tuned versions.

14.5.1   The ray tracer

As described in section 14.4 and in [3], the ray tracer consists of three major processes — `controller`, `calculator` and `graphicsEngine`.

14.5.2   The controller process

The controller is at the heart of the processor farm. The internal structure of the controller is illustrated below.
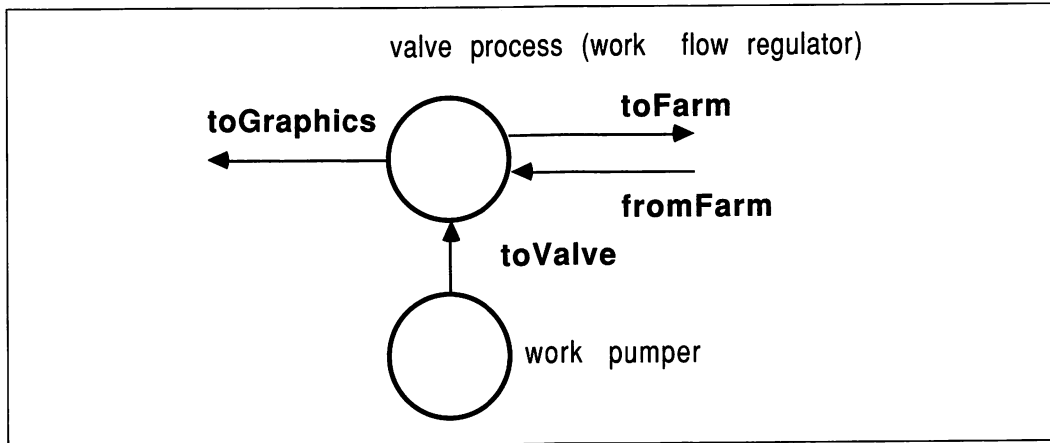

Figure 14.5 The controller process

The valve process is regulating the flow of work into the farm of calculators, and passing results packets on to the graphics card. It is very important that the controller responds quickly to incoming results packets. Therefore the process accepting results packets is prioritised, and the **ALT** construct in the valve process is prioritised to accept results rather than pass on work. Each calculator has a buffered work packet, so it is more important that results be passed on to the graphics card rather than more work packets passed out to the farm.

14.5.3   The calculator process

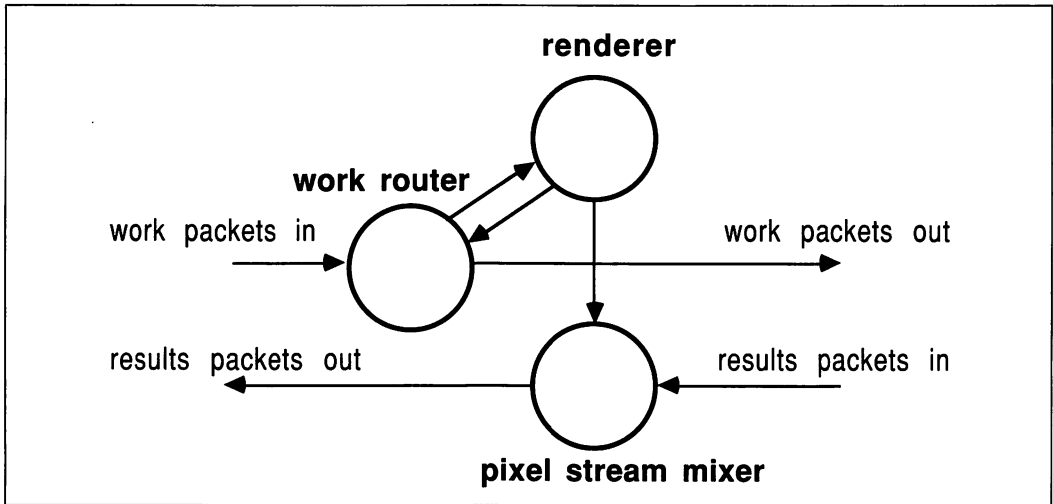The calculator contains a work router, a pixel stream mixer and a renderer (section 3.1).

Figure 14.6 The calculator process

All the vectors used by `mixPixels routeWork` and `calculate` are declared at the outermost lexical level, and passed into the processes as parameters. Keeping the workspace of the work routing processes in internal memory is very important in a processor farm, as the latency of response to link inputs is reduced. When a process is scheduled, several words are written into the workspace of the descheduled process, and these write cycles will be slower if the workspace is off-chip, thus increasing process-swap time and degrading the performance of the farm as a whole.

14.5.4   The graphics process

The graphics process accepts pixels from the controller and plots them on a IMS B007 graphics board [5]. The internal structure of the graphics process is illustrated below.
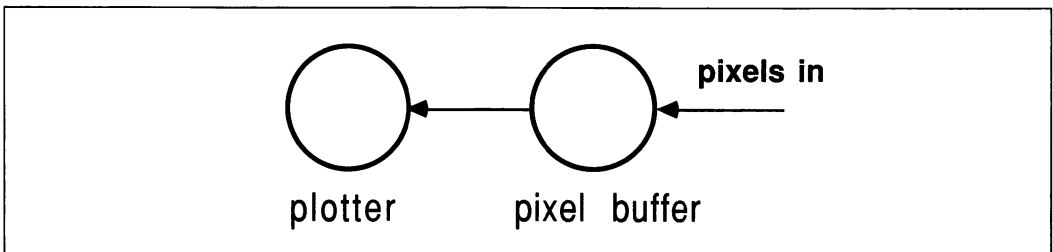

Figure 14.7 The graphics process

The buffer process in `graphicsEngine` improves overall performance slightly, by overlaying the plotting of one patch with inputting the next. The buffer process is prioritised over the plotter.

14.6   Conclusions

Several techniques have been presented for performance enhancement of occam programs running on transputers.

These techniques can be summarised as:

Enhancement technique	Section
Keep workspaces in on-chip memory	14.2.1
Use abbreviations to minimise static chaining	14.2.2
Use abbreviations to remove range checking	14.2.2
Use abbreviations to open out loops	14.2.2
Place critical vectors on-chip	14.2.3
Initialise large vectors with block move	14.2.4
Use retyping to accelerate byte manipulation	14.2.5
Use TIMES	14.2.6
Decouple communication and computation	14.3.1
Use buffer processes on links where necessary	14.3.1
Prioritise processes which use links	14.3.1
Keep messages as long as possible	14.3.2
Use dynamic load balancing if appropriate	14.4

Some techniques (dynamic load balancing, link buffering, buffer process prioritisation) are applicable only to arrays of transputers, others (optimum use of on-chip memory) should be applied at all times.

It has been shown that severe performance degradation can occur if an occam program is written without appropriate application of these techniques. Therefore these techniques should be considered for all occam applications.

14.7   Handling recursion in occam

occam does not allow recursion, so recursive algorithms must be restated in a non-recursive manner. A good example is the anti-aliasing algorithm from the ray tracer.

In computer graphics, *anti-aliasing* is a term used to describe algorithms which reduce perceptually disturbing artefacts in images. These artefacts are *aliases*, and are due to the point-sampling nature of computer graphics algorithms (see [10]). In order to reduce these aliases (and hence generate more realistic images) it is necessary to perform area-sampling, so that the colour assigned to each pixel on the display is an integration over the entire pixel area, rather than a single point sample.

The simplest approach to anti-aliasing is therefore to supersample each pixel (e.g. trace 16 rays rather than 1) and return the average colour — this implies a factor of 16 increase in the work load, over an already compute-intensive algorithm. Therefore an *adaptive supersample* is performed.

The purpose of adaptive supersampling is to generate an anti-aliased image without the expense of super-sampling all pixels in the image. The algorithm supersamples those pixels where detectable colour changes have occured, splitting these pixels into four sub-pixels and recurring. This results (in most cases) in an acceptable image at an average 30–50% increase in computation time over a simple ray trace.

Expressed recursively in PASCAL, the algorithm is

```
FUNCTION averageColour ( x0, y0, size, level : INTEGER) : INTEGER;
FORWARD;

FUNCTION averageColour { x0, y0, size, level : INTEGER) : INTEGER };
VAR
  A, B, C, D, half : INTEGER;
```
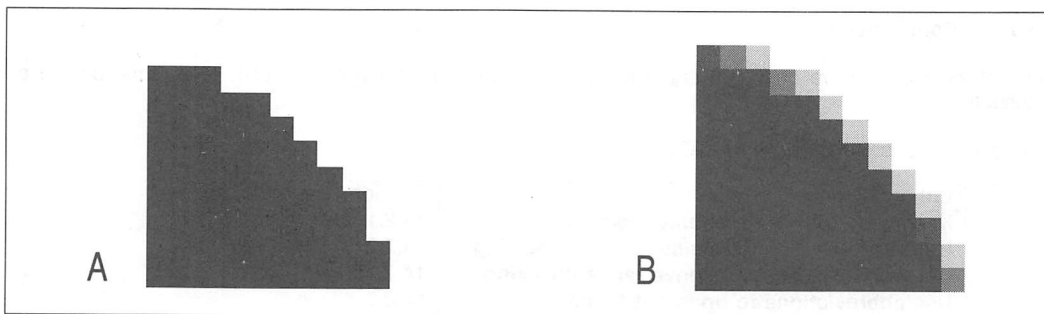
Figure 14.8 Magnified object silhoutte A) without and B) with anti-aliasing

```
BEGIN
  A := rayTrace ( x0, y0);
  B := rayTrace ( x0+size, y0);
  C := rayTrace ( x0, y0+size);
  D := rayTrace ( x0+size, y0+size);
  IF (level < maxLevel) AND
     (colourDifference ( A, B, C, D) > maxDiff) THEN
  BEGIN
    half := size / 2
    averageColour :=
             ( averageColour ( x0,       y0,       half, level+1) +
               averageColour ( x0+half, y0,       half, level+1) +
               averageColour ( x0,       y0+half, half, level+1) +
               averageColour ( x0+half, y0+half, half, level+1)) / 4
  END
  ELSE
    averageColour := (A + B + C + D) / 4;
END;
```

The recursion bottoms out either when a maximum recursion level has been reached, or when the colour difference across the corners of the pixel is deemed acceptable. The INMOS implementation has the maximum recursion level set to 2, so up to 16 rays will be traced per pixel for anti-aliasing.

In occam, the implementation is more verbose, but is simple to understand. The program explicitly manipulates 2 stacks — actions (i.e. what the program should do next) and parameters (i.e. the data on which the program shall act) are stored on one stack, and returned results (in this case colour values) are kept on the other.

An action value is popped off the stack and the appropriate action performed. If a *TRACE* action is to be performed then four points (representing the corners of the pixel) are raytraced, and their colours compared — if the colour spread is acceptable then the average colour is pushed onto the colour stack, otherwise a *MIX* action and four further *TRACE* actions are pushed onto the action stack.

If a *MIX* action is to be performed, four colour values are popped off the colour stack, and their average pushed back.

The algorithm terminates on a *HALT* action, at which time the pixel's colour is held on top of the colour stack.

```
PROC averageColour ( INT averageColour,
                     VAL INT x0, y0, size0 )
  ...   declare actions - HALT  MIX a b c d and TRACE x0 y0 size level
  ...   declare variables, declare stacks, sp
  ...   procs to manipulate action / parameter stack
  ...   procs to manipulate colour stack
```

```
SEQ
   ... init stack pointers
   push1Action ( HALT )          -- pre-load stack with HALT action
   push4Action ( x0,y0,size0,1)  -- and parameters for this pixel
   action := TRACE
   WHILE action <> HALT
     IF
       action = TRACE
         INT a, b, c, d, diff :
         SEQ
           pop4Action ( x, y, size, level )
           rayTrace   ( a, x, y )
           rayTrace   ( b, x+size, y )
           rayTrace   ( c, x, y+size )
           rayTrace   ( d, x+size, y+size )
           colourDifference ( diff, a, b, c, d )
           IF
              (level < maxLevel) AND (diff > maxDiff)
                SEQ
                  size  := size / 2
                  level := level+1
                  push1Action ( MIX )
                  push5Action ( TRACE,x,y,size,level )
                  push5Action ( TRACE,x+size, y,size,level )
                  push5Action ( TRACE,x,y+size,size,level )
                  push4Action ( x+size,y+size,size,level )
              TRUE
                SEQ
                  push1Colour (((a + b) + (c + d)) / 4)
                  pop1Action ( action)
       action = MIX
         INT a, b, c, d :
         SEQ
           pop4Colour  ( a, b, c, d )
           push1Colour (((a + b) + (c + d)) / 4)
           pop1Action ( action)
   pop1Colour ( averageColour)
:
```
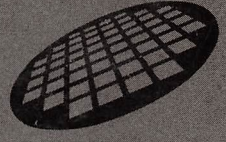
Note that as presented the algorithm is extremely inefficient, re-ray tracing points several times over. The algorithm as implemented caches previous results (in a large vector declared at the outermost lexical level and abbreviated into a local variable).

14.8   References

1 *Transputer reference manual*, Prentice Hall. 1988

2 *occam 2 reference manual*, Prentice Hall. 1988

3 *Exploiting concurrency; a ray tracing example*, Technical note 07, INMOS Limited, Bristol. 1987

4 *IMS B004 IBM PC add-in board*, Technical note 11, INMOS Limited, Bristol. 1987

5 *IMS B007 a transputer based graphics board*, Technical note 12, INMOS Limited, Bristol. 1987

6 *Signal Processing With Transputers* J.G. Harp, J.B.G. Roberts and J.S. Ward, Computer Physics Communications, January 1985

7 *Communicating Process Computers*, Technical note 22, INMOS Limited, Bristol. 1987

8 *Turbocharging Mandelbrot*, Dick Pountain, BYTE magazine, September 1986

9 *Evaporated film profiles over steps in substrates*, I.A Blech, Thin Solid Films, 6 (1970) pp 113–118

10 *An Improved Illumination Model For Shaded Display*, Turner Whitted,
   Communications Of The ACM, pp 343–349, June 1980, 23 (6).

Transputer Technical Notes is a collection of technical notes written by INMOS engineers. The collection will be of interest to electronic engineers, software engineers, programmers, system designers and managers. The collection is relevant to anyone currently using or thinking of using transputer technology, including INMOS communication devices. The following subjects are included:

- IMS T414 and IMS T800 memory interface.
- Connecting INMOS link adaptors.
- IMS B003 design of a multi-transputer board.
- Design and applications for the IMS C004 link switch.
- Module motherboard architecture.
- Dual inline transputer modules (TRAMs).
- Program design for concurrent systems.
- Exploring multiple transputer networks.
- Extraordinary use of transputer links.
- Transputer based radio-navigation system.
- Testing embedded sytems.
- Transputer based graphics display.
- Benchmarks.
- Performance maximisation.

⬤inmos

Prentice Hall